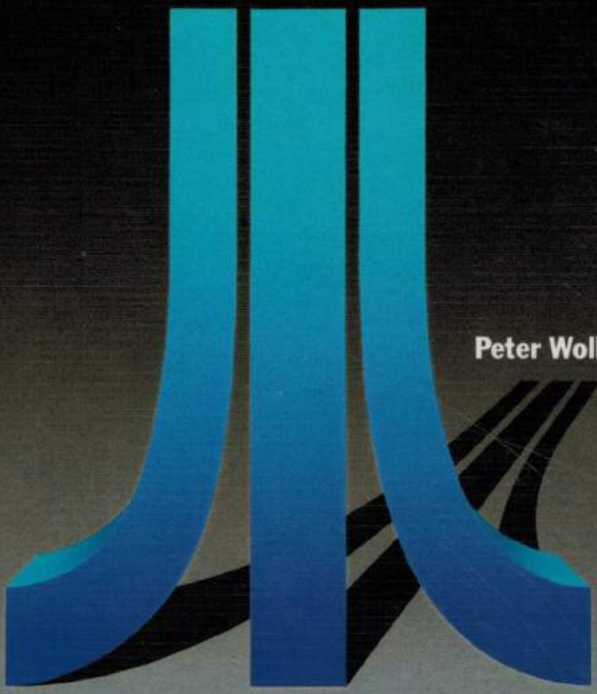
Markt&Technik



Peter Wollschläger

ATARI ST ASSEMBLER-BUCH

Ein 68000-Kurs mit vielen Beispielen.

Mit Tips für das Einbinden von Assemblerroutinen in Hochsprachen (BASIC, C) und ausführlichem Verzeichnis aller GEMDOS-, BIOS- und XBIOS-Funktionen.

Enthalten:
3½"-Diskette mit RAM-Disk-Programm, Disk-Monitor und anderen ST-Utilities.

Atari ST Assembler-Buch

Das vorliegende Buch erschien 1987 beim Markt&Technik Verlag unter der ISBN-Nummer 3-89090-467-X und ist aktuell in gedruckter Form nur noch als gebrauchtes Exemplar erhältlich.

Für meinen persönlichen Zweck hatte ich mir dieses Buch eingescannt um das Original in seinem jetzigen Zustand erhalten zu können. Da ich aber die Meinung vertrete das solche Bücher auch in digitaler Form für die Zukunft erhalten bleiben sollten, habe ich eine Anfrage zur Erlaubnis einer Veröffentlichung dieses PDFs beim heutigen Markt+Technik Verlag eingeholt.

Der Verlag hat am 11.04.2020 zugestimmt und somit kann ich dieses Werk der Allgemeinheit zur Verfügung stellen!

Der Rechte am ursprünglichen Markt&Technik Verlag wurden 2014 von der Braun Handels GmbH übernommen, dies inklusive allen früheren Buchreihen. Der Name „Markt&Technik“ wurde beibehalten, die Schreibweise lautet aktuell aber nun „Markt+Technik“.

Als Plattform zur Veröffentlichung dieses PDFs, habe ich das größte deutsche Atari-Forum ausgewählt:

http://www.atari-home.de

X-Ray / 12.April 2020

Peter Wollschläger

Atari-ST-Assembler-Buch

Ein 68000-Kurs mit vielen Beispielen. Mit Tips für das Einbinden von Assemblerroutinen in Hochsprachen (BASIC, C) und ausführlichem Verzeichnis aller GEMDOS-, BIOS- und XBIOS-Funktionen

Markt & Technik Verlag AG

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Wollschläger, Peter:
ATARI-ST-Assembler-Buch : e. 68000-Kurs mit vielen Beispielen, mit Tips für d. Einbinden von Assemblerroutinen in Hochsprachen (BASIC, C) u. ausführl. Verz. aller GEMDOS-, BIOS- u. XBIOS-Funktionen / Peter Wollschläger. – Haar bei München : Markt-und-Technik-Verlag, 1987.
ISBN 3-89090-467-X

Die Informationen im vorliegenden Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Buch gezeigten Modelle und Arbeiten ist nicht zulässig.

ATARI® ist ein Warenzeichen der Atari Inc., USA

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
90 89 88 87

ISBN 3-89090-467-X

© 1987 by Markt&Technik Verlag Aktiengesellschaft,
Hans-Pinsel-Straße 2, D-8013 Haar bei München/West-Germany
Alle Rechte vorbehalten
Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Kösel, Kempten
Printed in Germany

Inhaltsverzeichnis

Vorwort 11
Wer sollte dieses Buch wie lesen? 13

Kapitel 1
Assembler: Was, wie, wann und womit? 15

1.1 Ganz unten: Maschinensprache 17
1.2 Höher: Assemblersprache 17
1.3 Ganz oben: Hochsprachen 19
1.4 Assembler im Prinzip oder: Warum so umständlich? 19
1.5 Wann Assembler und wann besser nicht 20
1.6 Was man wofür an Software braucht 20
1.6.1 Der Editor 21
1.6.2 Der Assembler 21
1.6.3 Der Linker 22
1.6.4 Der Debugger 22
1.7 Was man kaufen und kann und sollte (und was nicht) 23
1.7.1 Der Digital-Research-Assembler 23
1.7.2 Der Kuma-Assembler 23
1.7.3 Der Metacomco-Assembler 23
1.7.4 Der GST-Assembler 24
1.7.5 Der Profimat 24
1.7.6 Der IDEAL-Assembler von Omnikron 24

Kapitel 2
Aufbau und Funktion eines Computers 25

2.1 Das Computer-Modell 27
2.2 Fetch & Execute 28
2.3 Programme sind nur Bytefolgen 28
2.4 Die Ausnahme ist die Regel 29

2.5 Das hexadezimale Zahlensystem 31
2.6 Ein Basic-Programm zum Üben 31
2.7 Das duale Zahlensystem 31
2.8 Der Sinn der Bomben und Pilze 33
2.9 Stack: Funktion und Aufgaben 33

Kapitel 3
Adressen, Daten und Befehle 37

3.1 Tempo durch Register 39
3.2 Das Register-Modell des 68000 40
3.3 Datentypen 40
3.4 Befehle 41
3.5 Sinn und Zweck der Adressierungsarten 42
3.6 Die Adressierungsarten im Detail 44
3.6.1 Register direkt 45
3.6.2 ARI: Adreßregister indirekt 45
3.6.3 ARI mit Postinkrement 45
3.6.4 ARI mit Predekrement 45
3.6.5 ARI mit Adreßdistanz 45
3.6.6 ARI mit Adreßdistanz und Index 46
3.6.7 Absolute Adressierung 46
3.6.8 Konstanten-Adressierung 46
3.6.9 PC-relative Adressierung 47

Kapitel 4
Ganz schnell zur Praxis 49

4.1 Ein Schnellkurs in Sachen TOS 51
4.2 Aufruf von TOS-Routinen 51
4.3 Form eines Assembler-Programms 51
4.4 Das erste Listing: Ausgabe von Zeichen 52
4.5 Assemblieren und Linken 54
4.6 Bequemer mit BATCH.TTP 55
4.7 »Print Hallo«, Version 2 55
4.8 Ausgabe von Strings 56
4.9 Typwandlung muß sein 57
4.10 Schleife mit »IF..THEN« 58
4.11 String-Ausgabe mit GEMDOS #9 60
4.12 Die erste DBcc-Schleife 61
4.13 Eingabe von Strings 63

4.14 Programm-Segmente Text, Data und BSS 64
4.15 Unterprogramme 65
4.16 String-Eingabe mit GEMDOS #10 66
4.17 Text ins String-Format 66

Kapitel 5
Verzweigungen und Menü-Technik 69

5.1 »IF THEN« im Detail 71
5.1.1 Das Statusregister 71
5.1.2 Die Flags 72
5.1.3 Die Abfrage der Flags 72
5.2 Bit-Schieben muß sein 73
5.2.1 Ein Hex-Konverter 74
5.2.2 Die Sache mit den Masken 75
5.3 ASCII-Code und Scan-Codes 76
5.4 Lesen der Funktions-Tasten 77
5.5 Die Mehrfachverzweigung 78
5.5.1 Lösung 1: mit vielen »IF THEN« 79
5.5.2 Lösung 2: ON X GOSUB in Assembler 79
5.5.3 Lösung 3: CASE X OF 82
5.6 Arbeiten mit zwei Tabellen 82
5.7 Der Location-Counter und Equates 85
5.8 Suchen mit DBcc 86

Kapitel 6
Das Betriebssystem im Detail 87

6.1 Wer macht was? 89
6.2 GEMDOS-Funktionen (Trap #1) 90
6.3 BIOS-Funktionen (Trap #13) 91
6.4 XBIOS-Funktionen (Trap #14) 92
6.5 GEM, VDI und AES 93

Kapitel 7
Rationalisierung der Arbeit 95

7.1 Strukturierung von Assembler-Programmen 97
7.1.1 Struktur in der Sprache 98
7.2 Makros 99

7.3 Include-Files 104
7.4 Module 104
7.4.1 Textmodule 105
7.4.2 Code-Module 105
7.5 Top Down, Bottom Up 107
7.6 Programm-Entwicklung am Beispiel »bindec« 108

Kapitel 8
Das File-System des Atari ST 119

8.1 Überblick 121
8.2 Das Directory 122
8.3 Directory-Entries 122
8.4 Der DTA-Puffer 124
8.5 Directory ausgeben 124
8.6 Boot-Sektor und BIOS-Parameter-Block 129
8.7 Ausgabe des BPB 130
8.8 FAT und Cluster (und halbe Bytes) 132
8.9 Einrichten, Schreiben, Lesen und Kopieren von Dateien 139
8.10 Der Overhead 144
8.10.1 Der Programmkopf 144
8.10.2 Die Base-Page 146

Kapitel 9
Schneller gehts nicht: Grafik in Assembler 149

9.1 Die Befehlsemulatoren des 68000 151
9.1.1 LINE-F-Emulator 154
9.2 Die Line-A-Grafik 154
9.3 Die Line-A-Variablen 155
9.4 Line-A-Grafik in der Praxis 159
9.4.1 Beispiel 1: Zeichnen von Linien 159
9.4.2 Beispiel 2: Zeichnen von Linien mit Trick 160
9.4.3 Beispiel 3: Animation 162

Kapitel 10
Die Befehle des 68000 im Überblick 169

10.1 Transfer-Befehle 171
10.1.1 LINK und UNLK 171

10.2 Arithmetische Befehle 172
10.2.1 BCD-Arithmetik 173
10.3 Logische Befehle 174
10.4 Bit-Befehle 175
10.5 Schiebe- und Rotierbefehle 175
10.6 Programmsteuer-Befehle 176

Kapitel 11
Der 68000 im Detail 179

11.1 Die innere Struktur des 68000 181
11.2 User- und Supervisor-Modus 182
11.3 Exceptions 183

Kapitel 12
Utility 1 187

12.1 High Speed: Ein RAM-Disk-Programm 189
12.2 Die vollautomatische RAM-Disk 200

Kapitel 13
Utility 2
Ein Disk-Monitor 203

13.1 Menü-Technik 214
13.2 Bildschirm-Gestaltung 215
13.3 Displays 215
13.4 Zugriff auf das XBIOS 216
13.5 Kopierschutz 216
13.6 Zahlenwandlungen 217

Kapitel 14
Einbindung von Assembler-Routinen in Hochsprachen 219

14.1 Basic 221
14.1.1 Modul nachladen 221
14.1.2 Routine in den RAM POKEn 221
14.2 Pascal 223
14.3 Die Sprache C 224

Kapitel 15
Die Tricks der Profis 227

15.1 So startet der ST 229
15.2 So muß ein Cartridge-Programm aussehen 230
15.3 Der Trap-Dispatcher des BIOS 234

Anhang 239

A1 Befehlsliste des 68000 241
A2 GEMDOS-Funktionen 255
A3 BIOS-Funktionen 265
A4 XBIOS-Funktionen 269
A5 LINE-A-Grafik 277
A6 Der Zeichensatz des Atari-ST 281
A7 Der VT52-Emulator 285
A8 Die Systemvariablen 289
A9 Liste der Exception-Vektoren des ST 293

Stichwortverzeichnis 295

Übersicht über weitere Markt & Technik-Produkte 300

Vorwort

Ich selbst habe seinerzeit Assembler auf einer IBM-360 gelernt. Damals war Rechenzeit noch unheimlich teuer, weshalb man uns Anfänger zuerst mit sehr, sehr viel Theorie traktiert hat, bevor wir den kostbaren Computer mit unseren ersten einfachen Programmen belästigen durften.

Während der vielen Stunden Theorie hatte ich immer Probleme, mir vorzustellen, wofür denn wohl was gut sein könnte. Richtig kapiert habe ich einiges von dem erst viel später, nämlich in der Praxis.

Diese Vorgehensweise trifft man leider auch noch in vielen Büchern an. Nach zum Teil Hunderten von Seiten nackter Theorie kommt da endlich mal das erste Programm, wenn Sie Pech haben, sogar erst in Band 2.

Ich möchte da anders vorgehen! Die Theorie soll nur so weit gehen, wie es für das Verständnis des ersten einfachen Programms unbedingt erforderlich ist. Leider sind wir dann schon beim vierten Kapitel angekommen, aber ganz ohne Grundlagen geht es halt doch nicht.

Wenn das erste Programm läuft, lesen Sie weiter Theorie, bis hin zum nächsten Programm, das dann schon etwas schwieriger ist. So arbeiten wir uns langsam hoch, bis zum Schluß einige nützliche Programme wie ein Disk-Monitor und eine RAM-Disk laufen und Sie dann genau wissen, wie so etwas funktioniert und wie Sie solche und bessere und noch kompliziertere Programme selbst schreiben können.

Noch etwas: Ein Assembler ist immer an eine ganz bestimmte CPU gebunden, hier an den 68000. Diese CPU gibt es zwar im Amiga oder im Macintosh auch, deshalb läuft aber ein Amiga-Programm noch lange nicht auf dem Atari.

Aus diesem Grund verzichte ich auf den ganz großen Leserkreis und schreibe dieses Buch speziell für den Atari ST. Trotzdem, wenn Sie mal auf einen anderen 68000er umsteigen, dann können Sie Ihr hier erworbenes Wissen mitnehmen, Sie müssen nur die Interna des Betriebssystems des anderen Rechners neu lernen.

Womit ich noch eine Aufgabe erwähnt hätte: Assembler-Programmierung ohne solide Kenntnisse von GEMDOS, BIOS und XBIOS ist nicht möglich. Auch das ist also nicht nur ein Kapitel in diesem Buch, sondern ein Faden (oder ein Seil), der sich durch alle Kapitel zieht. Zum Schluß ein guter Rat: Wenn ein Programm nicht läuft, dann sind immer die berühmten Kleinigkeiten die Ursache. Hübsch häßlich ist nun leider die Tatsache, daß in Assembler jeder noch so kleine Fehler mit kräftigen Strafzeiten geahndet

wird, weil nach der Fehlerbeseitigung kein einfaches RUN reicht, sondern einige Programme abgearbeitet werden müssen.

Lassen Sie sich dadurch nicht entmutigen! Mit jedem Fehler lernen Sie dazu, und schließlich machen Sie immer weniger Fehler. Eine gewisse Zähigkeit gehört allerdings dazu. Andererseits: Wer eine Programmiersprache gelernt hat, und ich unterstelle, Basic oder etwas anderes können Sie schon, der lernt auch eine zweite Sprache. Generell ist Assembler nicht schwieriger als Basic, nur leider etwas umständlicher. Zu diesen Umständlichkeiten gehört auch, daß Sie – im Gegensatz zu Basic – hier wissen müssen, wie der Computer funktioniert.

Aber irgendwie macht es mir auch mehr Spaß, wenn ich meinen Computer sozusagen direkt programmieren kann. In einer Hochsprache bin ich immer auf die Güte (auch im Sinne von Gnädigkeit) des Compilers oder Interpreters angewiesen. Wenn das Ding ungenau rechnet oder viel zu langsam ist, ist mit dieser Erkenntnis die Sache ungelöst beendet, es sei denn, man kann in Assembler das Problem nun richtig angehen.

In diesem Sinne (und nicht aufgeben!!)

Ihr

Peter Wollschlaeger

Wer sollte dieses Buch wie lesen?

Das Buch wendet sich an Einsteiger und Umsteiger. Letztere können im Kapitel 1 die Abschnitte 1.1 bis 1.7 überspringen, im Kapitel 2 die Abschnitte 2.1 und 2.2, und das war's auch schon.

Ich muß die Umsteiger, soweit sie von den »8-Bittern« kommen, leider enttäuschen. Vorkenntnisse vom Z80, 8088 oder 6502 her nützen beim 68000 herzlich wenig, noch schlimmer, sie könnten sogar stören. Ernsthaft: Als Z80- oder 6502-Programmierer gewöhnt man sich gewisse Techniken und Denkweisen an, die zwar auf den 68000 übertragbar sind, aber dann nur unnötig lange Programme ergeben. Vergessen Sie alle Adressierungsarten, die Sie da gelernt haben, den Begriff Akku streichen Sie ganz, Banking und Paging erst recht und noch vieles mehr. Am besten, Sie vergessen alles!

Haben Sie schon 68000-Erfahrung, dann können Sie bei Kapitel 4 beginnen. Ähnliches gilt für die Leser, die von den Minis her kommen, speziell von der VAX.

Der 68000 ist zwar ein Prozessor, über dessen fantastische Eigenschaften man bücherfüllend hochgestochene Traktate für Diplom-Informatiker verfassen kann, aber genau das tue ich nicht. In den Kapiteln 1 bis 3 bringe ich nur die Grundlagen, die Sie kennen sollten, um die ersten Programme schreiben zu können. Dann folgt Praxis, Praxis, Praxis. Erst in den Kaptiteln 10 und 11 beginnt die Würdigung des 68000, und dann folgt schon wieder Praxis.

Während des Praktikums wird folgende Linie verfolgt.

1. Schilderung der Aufgabe
2. Vorstellung der dafür erforderlichen Befehle und TOS-Funktionen
3. Das Programmlisting
4. Die Erklärung des Listings

Stellenweise wird diese Ordnung durchbrochen, weil es manchmal sinnvoller ist, den Punkt 2 im Zusammenhang mit dem Listing zu erklären.

Auf jeden Fall sollten Sie niemals zuerst das Listing lesen, sondern es im ersten Ansatz überspringen.

Die Fülle der Informationen aller Kapitel sich zu merken, dürfte schwierig sein. Deshalb sind in den Anhängen unter anderem die 68000-Befehle und die TOS-Funktionen in einem kompakten Format zusammengefaßt. Darüber und mittels des Stichwort-Verzeichnisses sollte ein schnelles Nachschlagen möglich sein.

Kapitel 1

Assembler:

Was, wie, wann und womit?

Was ist Assembler?

Wie programmiert man in Assembler?

Wann braucht man Assembler?

Was braucht man an Software?

In diesem Kapitel soll zuerst einmal gezeigt werden, was Assembler ist, wann man ihn braucht und was man an Software benötigt, um ein Programm in Assembler erstellen zu können.

Eines vorab: Wenn mal ein paar Fachausdrücke auftauchen, die Sie nicht verstehen, einfach weiterlesen. Wenn wir sie wirklich brauchen, werden sie auch erklärt.

1.1 Ganz unten: Maschinensprache

Ein Computer an sich ist sehr dumm, nur sagenhaft fleißig. Gehässige Leute sagen auch, nur wer so dumm ist, ist auch so fleißig. Tatsächlich kann diese Maschine nicht bis drei zählen, noch nicht einmal bis zwei, sie kennt gerade die Null und die Eins. Ursache ist, daß die elektrischen Schaltkreise, aus denen ein Computer besteht, nur zwei Zustände annehmen können, nämlich »Spannung da« oder »Spannung nicht da«, Strom fließt oder fließt nicht, ein Transistor leitet oder sperrt. Ein paar Hunderttausend dieser Schaltkreise (Transistoren) bilden nun die CPU (Central Processing Unit, Zentraleinheit, praktisch das Herz des Computers), nochmals mehr als 8 Millionen davon (so Sie einen 1040 ST haben) sind der Speicher (das Gedächtnis) des Rechners.

Ein Programm ist nichts weiter als ein bestimmter Zustand dieses Speichers. Da es nun höchst unpraktisch ist, ein Programm in der Art zu beschreiben »Transistor 1 leitet, Transistor 2 auch, Transistor 3 sperrt, Transistor 4 leitet usw.«, kam man schnell auf eine Kurzschreibweise dieser Art: Der eine Zustand heißt 0 der andere 1. So kann man ein Programm doch schön kompakt schreiben, zum Beispiel als:

```
0101110011010101010 usw.
```

Das gefällt Ihnen nicht? Nun, das ist die Maschinensprache, mehr nicht !!! Was Sie wohl schon erkannt haben: dieses 0101011-Muster ist eine Zahl in dualer Schreibweise (ich komme noch darauf zurück). Diese Zahlen kann man umrechnen in Dezimal- oder Hexadezimal-Zahlen, das spart etwas Papier, es bleibt aber Maschinensprache.

1.2 Höher: Assemblersprache

Manche Leute behaupten nun, Assembler sei diese Maschinensprache. Gott sei Dank haben die unrecht, das wäre ja schrecklich. Die armen Kerlchen, die die ersten Computer so programmiert haben, tun mir heute noch leid. Assembler ist die nächsthöhere Stufe und war einst der ganz große Fortschritt und viele Jahre lang auch die einzige Sprache überhaupt. Nun muß ich leider doch noch etwas ausholen.

Bits und Bytes

Wenn Sie bei Bit nicht mehr zuerst an Bier denken, sind Sie schon Programmierer, O.K. ... Ein Bit ist eine Speicherstelle, ein solcher Schaltkreis im Computer, der nur die Zustände 0 oder 1 annehmen kann. Aus technischen Gründen hat man immer 8 Bit zusammengefaßt, diese 8 Bit nennt man ein Byte. Der Speicher eines Computers besteht aus Tausenden oder Millionen von Bytes. Damit man nun jedes Byte ansprechen kann, sind sie durchnumeriert. Diese Hausnummern der Bytes nennt man Adressen. Mit den 8 Bit

eines Bytes lassen sich in dualer Schreibweise die Zahlen 00000000 bis 11111111 darstellen, dezimal ist das 0 bis 255. In ein Byte (der Fachmann sagt, auf eine Adresse) kann ich nun eine solche Zahl schreiben und sie wieder herauslesen. Die sog. Peripherie-Geräte – wie der Bildschirm, die Tastatur oder ein Drucker – sind nun mit einem Teil des Speichers (unseren Bytes) verbunden. Wenn ich auf die richtige Adresse eine Zahl schreibe, dann erzeugt sie eine Wirkung auf dem Bildschirm, wenn ich aus einer anderen Adresse etwas lese, dann kann das zum Beispiel eine Taste des Keyboards sein.

Bewegen ist alles

Folglich besteht ein Programm zum großen Teil daraus, Zahlen – man spricht auch von Daten – auf eine Adresse zu schreiben, von einer anderen zu lesen und, ganz wesentlich, Daten von einer Adresse (zum Beispiel Tastatur) auf eine andere Adresse (zum Beispiel Bildschirm) zu kopieren.

Neben den Daten kennt so ein Computer auch Befehle, natürlich auch nur als 010101110, sprich als Zahlen. Nehmen wir an, die Zahl 11111111 ist der Befehl »Kopiere« und wir wollen Daten von der Adresse 0000011 (dezimal 3) auf die Adresse 00001001 (dezimal 9) kopieren, dann lautet dieses Programm in Maschinensprache

```
11111111
00000011
00001001
```

In Assembler hingegen schreibt man dafür

```
MOVE 3,9
```

Move heißt bewege, hier: bewege, was im Byte mit der Adresse 3 steht zum Byte mit der Adresse 9. Um gleich einen großen Denkfehler auszuschließen: Das Byte 3 bleibt unverändert, es wird nur in das Byte 9 kopiert. Sie haben recht, der Befehl müßte eigentlich COPY heißen, aber er heißt nun mal MOVE.

So, den Unterschied zwischen Assembler und Maschinensprache hätten wir, ist doch ein Fortschritt, oder?

Doch schon hätten wir das nächste Problem. Der Begriff »Assembler« hat nämlich eine doppelte Bedeutung. Zum einen ist damit eine Programmiersprache gemeint, genauso wie zum Beispiel Basic oder Pascal. Der Unterschied ist hauptsächlich, daß Assembler immer an eine bestimmte CPU gebunden ist. Es gibt beispielsweise den Z80-Assembler, den 8088-Assembler und natürlich den 68000-Assembler, um den es hier geht. Die Sprache hat Befehle, wie alle anderen Sprachen auch, die tippen Sie auch einfach so ein, wie üblich. Der große Unterschied zu Basic zum Beispiel ist dann nur, daß Sie danach nicht RUN geben können, sondern den Text erst assemblieren müssen. Genau das erledigt ein Programm, das dummerweise auch Assembler heißt. Dieses Programm übersetzt den Text in die Maschinensprache, also die 0101010-Folge, die die CPU letztendlich nur versteht.

1.3 Ganz oben: Hochsprachen

In einer Hochsprache, wie zum Beispiel Pascal, geben Sie auch nur Text ein, auch der muß übersetzt werden, nur heißt dann das Übersetzungsprogramm nicht Assembler, sondern Compiler. Das heißt, sowohl nach einem Assembler- als auch nach einem Compilerlauf entsteht ein Programm in Maschinensprache, das auf einem Computer ausgeführt werden kann. Über Größe und Schnelligkeit der Programme ist damit noch nichts gesagt.

Ganz anders sieht es bei einem Interpreter aus; der typischste Vertreter dieser Gattung ist wohl Basic. Auch hier geben Sie das Programm als Text ein. Vielleicht wird es nach der Eingabe noch etwas aufbereitet und komprimiert, aber es bleibt Text, der nicht die geringste Ähnlichkeit mit Maschinensprache hat. Folglich kann der Computer ein Basic-Programm auch nicht direkt ausführen. Diese Aufgabe übernimmt der Interpreter. Er liest den Basic-Text Zeichen für Zeichen und untersucht ihn auf Basic-Befehle. Findet er einen Basic-Befehl, so ruft er eine Routine auf, die den Befehl ausführt. Die Routine befindet sich natürlich als ausführbares Maschinenprogramm im Speicher. Sie übernimmt es auch, zu einem Basic-Befehl gehörige Daten (Parameter) im Basic zu suchen. Selbstverständlich ist auch der Interpreter selbst ein Programm in Maschinensprache. Alle schnellen Basic-Interpreter sind in Assembler geschrieben.

1.4 Assembler im Prinzip oder: Warum so umständlich?

Ja, wenn nun der Compiler genauso Maschinen-Code erzeugt wie ein Assembler, dann sollte ich mir die Sache doch noch einmal genau überlegen. In Pascal zum Beispiel schreibe ich einfach nur

```
Write('Hallo')
```

und in Assembler tippe ich dafür (nur als Beispiel):

```
MOVE #'H',4711
MOVE #'a',4712
MOVE #'l',4713
MOVE #'l',4714
MOVE #'o',4715
```

Demnach ist ein Assembler-Programm die Auflösung von zum Beispiel Pascal-Befehlen wie WRITE in viele Einzelbefehle. Man kann es auch anders sagen: Pascal kennt eine bestimmte Menge von Befehlen, aus denen der Compiler die passende Folge von Assembler-Befehlen erzeugt. Tatsächlich ist jedes Assembler-Programm (in der noch nicht übersetzten Textform) immer länger als sein Äquivalent in einer Hochsprache. Nur wenn Sie einmal nach dem Assemblieren bzw. dem Compilieren jeweils die Bytes des Code zählen, dann ist ein Assembler-Programm drastisch kürzer und schon deshalb auch schneller. Das liegt daran, daß kein Compiler einen so kompakten Code generieren kann, wie es ein Assembler-Programmierer tut. Letzterer weiß ja, was er will, er kann jede Befehlsfolge »maßschneidern«, ein Compiler hingegen muß Universallösungen einsetzen.

Ganz drastisch, im Tempo so bis zu Faktor 200, ist der Unterschied zu einem Basic-Interpreter. Dieser übersetzt – wie schon geschildert – erst während der Laufzeit und dann immer nur einen Befehl. D. h., wenn in einer Schleife ein Befehl hundertmal wiederholt wird, dann wird er auch hundertmal übersetzt. In einem Assemblerprogramm hingegen ist der Befehl schon übersetzt.

1.5 Wann Assembler und wann besser nicht

Nun mag ja manche Leute das Tempo nicht stören, Sie haben Zeit, aber es gibt da noch einige Gründe für Assembler:

Ein Basic-Interpreter (oder ein Pascal-Compiler) kratzt eigentlich nur an der Oberfläche eines Riesenpotentials von Möglichkeiten, die in so einem Computer stecken. Will man mehr oder etwas anderes, dann muß man das der CPU nur sagen, allerdings in ihrer Sprache, und das ist nun mal Assembler.

Noch ein Grund: Man sollte eigentlich immer die Sprache verwenden, die das jeweilige Problem mit minimalem Aufwand löst. Oft genug, sogar meistens, ist das nicht Assembler. Ich möchte sogar fast behaupten, je besser man Assembler kann, desto weniger braucht man ihn. Ein Assembler-Programmierer weiß nämlich, was er mit welchen zum Beispiel Basic-Befehlen der CPU an Arbeit zumutet, und kommt so zwangsläufig zu besseren Programmen; denn das muß ich nun leider auch noch erwähnen: Assembler setzt gute Kenntnisse der Funktion eines Rechners voraus.

Aber zum Trost: diese Kenntnisse erwirbt man am besten, wenn man Assembler lernt.

Untersucht man nun ein Programm, das in einer Hochsprache geschrieben wurde oder geschrieben werden soll, dann stellt man fest, daß es nur an einigen Stellen (meistens nur an einer Stelle) das Tempo-Problem gibt oder die passende Funktion fehlt. Dann sollte man auch nur diesen Teil in Assembler schreiben und ihn in die Hochsprache einbinden. Wie das geht, wird später (genau: im Kapitel 14) erklärt.

Wie auch immer: Die Sprachen weit weg von der Maschine nennt man Hochsprachen, in Assembler sind wir »ganz unten«. Auch wenn Sie später nur noch in den höheren Regionen schweben, Sie wissen, mit einer soliden Grundausbildung schwebt es sich leichter, und man fällt nicht so leicht herunter.

1.6 Was man wofür an Software braucht

Die typische Arbeitsfolge einer Programmentwicklung in Assembler sind Texteingabe, Assemblieren, Linken (kommt gleich) und Testen. Das sind Ihre Werkzeuge, und wie in jedem Handwerk kommt es darauf an, daß Sie mit den richtigen Werkzeugen arbeiten. Da gibt es nun leider eine große Auswahl, und die Prospekte der Hersteller versprechen alle viel. Ich möchte Ihnen hier einige Tips geben, die Sie bei der Auswahl beachten sollten, und dann einige typische Erzeugnisse vorstellen. Vergessen Sie eines nie: Ein Assembler ist ein Profi-Werkzeug, das eine gute Dokumentation und Support braucht. Natürlich bekommen Sie ein (versehentlich) kopiertes Spielprogramm auch per »Trial and Error«

zum Laufen. Dies auf einen Assembler anzuwenden, dürfte nur etwas für Leute mit sehr guten Nerven und unendlich viel Zeit sein. Es kann nämlich durchaus sein, daß alle Programme in diesem Buch mit Ihrem speziellen Assembler nicht laufen, weil Ihr Assembler an einer Stelle einen Punkt verlangt, den meiner nicht braucht. Warum also knobeln, wenn alles im Handbuch steht.

1.6.1 Der Editor

Den Editor brauchen Sie für die Texteingabe und deren Korrektur. Den Text nennt man Quell-Text (Source-Text). Üblicherweise werden Editor und Assembler zusammen verkauft. Sie können aber auch ohne weiteres Ihr gewohntes Textverarbeitungs-Programm nehmen, wenn Sie sich auf reinen Text (keine Formatier- und Steuerzeichen) beschränken, bei »First Word« also den WP-Modus ausschalten. Der Assembler meldet Ihnen Fehler mit einer Zeilennummer, der Text wird aber ohne Zeilennummern eingegeben. Demnach sollte der Editor ein »Go To Zeile« können. Da Sie Programme ggf. umgestalten und recht oft Textteile kopieren (und geringfügig ändern), sollte das Bewegen und Kopieren von Blöcken möglich sein.

1.6.2 Der Assembler

Ist der Text fertig (und auf der Diskette), starten Sie den Assembler, der dann mindestens wissen will, wie das File mit dem Quelltext (Source-File) heißt. Der Assembler erzeugt den Maschinen-Code (dieses 010101010) – auch Objekt-Code genannt – und legt diesen in dem Ziel-File (dem Objekt-File) auf der Diskette ab. Das kostet natürlich Zeit, und so erscheint es sinnvoll, auch »in memory« assemblieren zu können. D. h., der Assembler schreibt auf Wunsch den Code direkt in den Speicher, und man kann das Programm zu Testzwecken starten. Dies Feature sollten Sie aber nicht überbewerten, denn das gleiche Ziel erreichen Sie auch mit einer RAM-Disk oder vom Zeitverhalten her gesehen auch mit einer Festplatte. In beiden Fällen muß natürlich der Assembler auch auf einer RAM-Disk bzw. einer Festplatte laufen. Darauf sollten Sie aber bestehen. Sonst sind noch folgende Eigenschaften wichtig:

»Include-Files«: Der Assembler kann Textmodule einbinden. Das ist sehr wichtig, denn nach einer Weile haben Sie sicherlich eine kleine Bibliothek von Routinen, die Sie in fast jedem Programm brauchen.

Makrofähig: Ausführlich werden Makros im Kapitel 7 behandelt. Hier nur soviel: Makros tragen stark zur Rationalisierung der Arbeit bei und helfen, Fehler zu vermeiden. Fehlermeldungen: Schauen Sie im Handbuch nach. Je länger die Liste der Fehlermeldungen, desto besser werden Sie informiert. Warnungen: Ein guter Assembler warnt Sie (berät Sie), wenn Sie nicht optimal programmiert haben. Auch hier gilt: Je mehr »Warnings«, desto besser.

1.6.3 Der Linker

Nun brauchen Sie den Linker, zu deutsch Binder. Der Binder hat zwei Aufgaben. Zum einen können Sie ein Programm in Module aufteilen, die Sie getrennt assemblieren und testen können (bei großen Programmen sehr empfehlenswert). Diese Module müssen Sie dann mit dem Linker zu einem Programm zusammenbinden. Der zweite Grund liegt beim Atari selbst. Jedes Programm hat einen kleinen Vorspann, Header genannt, in dem zum Beispiel steht, wie groß das Programm ist. Ohne diese Information kann das TOS das Programm nicht laden und starten. Folglich muß der Linker zumindest diesen Header mit Ihrem Programm binden. Es gibt aber auch Assembler, die das schon tun. Das erspart den Linkerlauf, was man durchaus positiv sehen sollte. Der Nachteil der fehlenden Modulisierung sollte dann aber durch »Include«-Fähigkeit und Makros ausgeglichen werden können.

1.6.4 Der Debugger

Starten Sie nun Ihr Programm, gibt es drei Möglichkeiten: entweder es läuft, oder es läuft nicht, oder es läuft falsch. Um den Bug (Programmierer-Slang für Fehler) zu finden, bieten sich viele Lösungen an. Die einfachste (und meist erfolgreichste) Methode ist ein tiefer Blick auf den Quelltext, kombiniert mit intensivem Nachdenken. Wenn Sie aber wissen wollen, was das Programm an einer bestimmten Stelle tut oder welche Werte dann einige Variable haben, wird es schwierig. Möglich ist es, an diesen Stellen sozusagen ein »PRINT A,B« einzubauen, was aber in Assembler recht aufwendig ist, wie wir noch sehen werden (es gibt keinen Print-Befehl). Praktischer ist es dann, einen sog. Debugger (Entwanzer, Fehler sind Wanzen!) einzusetzen. Das ist ein Programm, mit dem Sie Ihr Programm in Einzelschritten ablaufen lassen und sich an jeder Stelle die Werte der Variablen ansehen können. Versprechen Sie sich aber nicht zuviel von einem Debugger. So ein Programm ist gar nicht so einfach zu bedienen und wird Sie gerade in der Anfangsphase mehr verwirren, als daß es Ihnen hilft. Hinzu kommt, daß Beginner meistens Fehler begehen, die der Compiler schon findet. Nochmals, weil es so wichtig ist: Der Fehler steckt immer im Quelltext. Ein tiefer Blick darauf und intensives Nachdenken ist der beste Debugger! Wenn Sie einen Debugger erwerben, so müssen Sie auf zweierlei achten. Zuerst sollte es ein symbolischer Debugger sein. Das bedeutet folgendes: Im Assemblerprogramm arbeiten Sie niemals mit absoluten Adressen, sondern mit Labels (Marken), das sind dann die symbolischen Adressen. Der Assembler führt nun eine Tabelle, in der er zu den »Symbolen« die echten Zahlen notiert. Ein symbolischer Debugger greift nun einfach auf diese Symboltabelle des Assemblers zu. Daraus folgt nun die zweite Forderung, nämlich daß der Debugger das auch kann, sprich zum Assembler kompatibel ist.

1.7 Was man kaufen kann und sollte (und was nicht)

Editor, Compiler, Linker und Debugger (soweit vorhanden) werden meistens im Paket angeboten (so sollte es sein). Häufig gehört dazu noch eine sog. Shell (eigene Benutzeroberfäche), die es Ihnen gestattet, zum Beispiel direkt vom Editor in die Shell zu wechseln, wo Sie dann den Linker aufrufen. Das ist, sofern man ohne RAM-Disk arbeitet, schneller als der Umweg über den Schreibtisch. Ansonsten hat eine gute Shell den Vorteil, daß sie sozusagen ein für die Programmierung maßgeschneiderter Schreibtisch ist.

1.7.1 Der Digital-Research-Assembler

Dieser Assembler ist Teil des recht umfangreichen Entwicklungspakets von Atari für den ST. Obwohl der Assembler gut und schnell ist, würde ich nur wegen des Assemblers das ganze Paket nicht kaufen, weil Sie dann nämlich 5- bis 10mal soviel bezahlen wie für die anderen Assembler. Sollten Sie sich anders entschieden haben, kann ich Ihnen nur noch ST-Pascal empfehlen (die Version ohne Plus), denn dazu gehört FASTLINK, ein sehr schneller Linker. Der Linker von DR erzeugt nämlich nur ein Zwischenprodukt, das man mit einem weiteren Programm (RELMOD) nachbehandeln muß.

1.7.2 Der Kuma-Assembler (K-SEKA)

Von der Idee her ist der Assembler gut. Editor und Assembler sind gemeinsam im Hauptspeicher; wenn man es so nennen will, auch ein Linker. Assembliert wird »in memory«. Doch leider läßt die praktische Ausführung viele Wünsche offen. Der Editor ist sehr primitiv (Ein-Zeilen-Editor), und die üblichen Assembler-Direktiven nach Motorola-Standard wurden durch ganz andere Kürzel ersetzt. Zum Paket gehört ein eingebauter Debugger.

1.7.3 Der Metacomco-Assembler

Der Metacomco-Assembler ist guter Durchschnitt bis sehr gut, je nachdem, ob Sie die ältere Version (bis 10.200) oder die neuere (ab 10.300) mit eigener GEM-Shell nehmen. Er ist makrofähig, kann dann auch lokale Labels, und auch das »File-Including« ist erlaubt. Weil dieser Assembler zuverlässig arbeitet und sich an den Motorola-Standard hält, habe ich alle Beispiele in diesem Buch mit dem Metacomco-Assembler geschrieben und getestet. Der Bildschirm-Editor wird ohne Maus bedient und ist sehr kompakt und schnell. Für mich auch ein Vorteil: Der Assembler kann sowohl DR-Format (Standard) als auch GST-Format (sehr verbreitet) erzeugen. Ich muß allerdings gestehen, daß ich den Metacomco-Assembler zusammen mit meinem Lieblings-Editor (dem aus Megamax-C) und dem Linker (FASTLINK) des ST-Pascals einsetze.

1.7.4 Der GST-Assembler

Der GST-Assembler arbeitet mit eigener GEM-Shell, d. h. alle Programme können mit der Maus aufgerufen werden. Sein Editor ist eine Spar-Version von 1ST-Word, bei der ich besonders lästig finde, daß die File-Select-Box immer einen falschen Pfadnamen angibt (2 Schrägstriche hintereinander). Der Makroteil ist von extremer Leistungsfähigkeit. FOR-NEXT-Schleifen, WHILE-WEND-Konstrukte und vieles mehr werden in der »Mac-Lib« mitgeliefert, im Handbuch aber miserabel erklärt. Überhaupt ist das Handbuch die Schwachstelle. Wenn Sie es nämlich verstehen wollen, müssen Sie schon sehr gut Assembler können. Unerklärlich ist für mich, warum GST vor den Direktiven Text, Data und BSS das Wort SECTION fordert und damit vom Standard abweicht (mehr als drei Sections [»kriegen wir später«] kann das TOS sowieso nicht handhaben).

1.7.5 Der Profimat

Mit den Assemblern von DR, Metacomco und GST habe ich praktische Erfahrung. Den KUMA-Assembler habe ich einmal für »Computer persönlich« getestet und dann nie wieder angefaßt. Den Profimat habe ich bei einem Freund kurz bedient und fand ihn gut. Ich würde sagen, wo bei Kuma die gute Idee im Ansatz steckengeblieben ist, hat man bei Data-Becker zu Ende entwickelt. Der Assembler läuft wie GST unter einer GEM-Shell, der Editor auch, ein Linker fehlt, aber ein guter Debugger inkl. Disassembler gehört dazu. Der Assembler kann »in memory« assemblieren und ist makrofähig.

1.7.6 Der IDEAL-Assembler von Omnikron

Auch diesen Assembler habe ich für »Computer persönlich« getestet und dann nie wieder benutzt.

Um es kurz zu machen: Das Paket hat in etwa die Leistungsfähigkeit des Profimat, aber ohne GEM-Shell und – deshalb finde ich IDEAL nicht ideal – ohne Makros.

Kapitel 2

Aufbau und Funktion eines Computers

Register

Stapel

2.1 Das Computer-Modell

In diesem Kapitel müssen wir uns etwas mit dem Aufbau und der Funktion eines Computers beschäftigen. Wie das System elektrisch funktioniert, ist dabei aber völlig uninteressant. Für die Programmierung reicht immer ein sogenanntes Modell. Sie müssen wissen, was CPU, RAM, ROM und Bus bedeuten und leisten. Innerhalb der CPU interessiert dann ganz besonders das Register-Modell.

Sie sind in der Situation eines angehenden Autofahrers. Ich erkläre Ihnen jetzt, welchen Sinn Lenkrad, Kupplung, Gaspedal und Bremse haben.

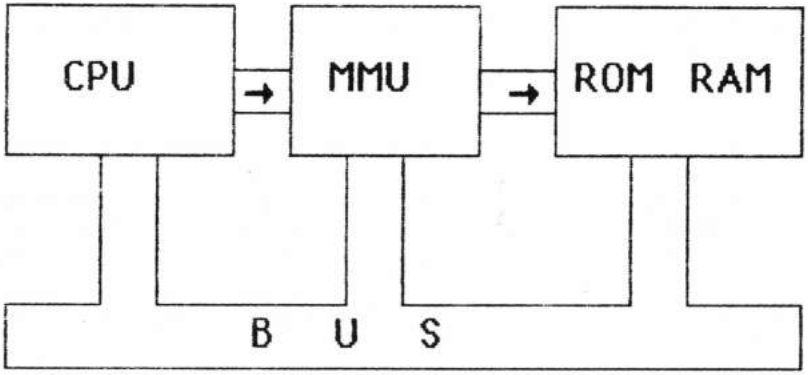


Bild 2.1: Modell eines Computers

Bild 2.1 zeigt das Modell eines Computers ziemlich vereinfacht. Der wichtigste Baustein ist die CPU (Central Processing Unit) oder der Prozessor, also der 68000. Er ist für die gesamte Ablaufsteuerung verantwortlich, er kann rechnen, entscheiden und vergleichen. Von allein tut er allerdings nicht viel, er braucht dafür ein Programm. Der zweite große Baustein ist der Speicher, unterteilt in die Teile RAM und ROM. RAM heißt historisch Random Access Memory, also Speicher mit wahlfreiem Zugriff (man kann direkt auf jede Speicherstelle zugreifen und nicht wie zum Beispiel beim Bandspeicher nur seriell). Nur – diese Eigenschaft hat ein ROM auch. Der große Unterschied: beim RAM ist Lesen und Schreiben möglich, beim ROM (Read Only Memory) nur Lesen. Noch ein Unterschied: Der RAM-Inhalt ist verloren, wenn Sie den Computer ausschalten, der ROM-Inhalt ist permanent vorhanden.

Unsere Programme werden immer im RAM liegen, wir werden aber den ROM-Speicher (so Sie einen ST mit ROM-TOS haben) kräftig nutzen. Um eine Speicherstelle im RAM oder ROM ansprechen zu können, muß die CPU diese Speicherstelle adressieren. Diese

Adressen laufen über den Adreßbus. Der Bus ist nichts weiter als eine Menge von Leitungen, über die alle Teilnehmer parallel geschaltet sind. Der Ausdruck Bus kommt daher, weil bildlich gesehen eine Information (zum Beispiel Adresse) an der Haltestelle CPU einsteigt, auf dem Bus fährt und dann an der Haltestelle RAM (oder ROM) aussteigt. Sinngemäß laufen die Daten (was in die adressierten Bytes hinein soll/aus ihnen gelesen wird) über den Datenbus.

Der Speicher selbst besteht aus vielen gleichartigen Chips. Diese haben alle den gleichen kleinen Adreßbereich, aber auch einen Eingang (Chip Select), über den man einen Chip anwählen kann. Deshalb muß eine logische Adresse in eine physikalische Adresse umgesetzt werden. Dies macht ein sog. Adreß-Dekoder oder – wie beim ST – eine leistungsfähigere Version davon, die sog. MMU (Memory Management Unit). Für uns ist wichtig zu wissen, daß durch Verschaltungen von CPU und MMU (also per Hardware) dafür gesorgt ist, daß ein Zugriff auf geschützte Bereiche oder illegale Adressen mit einem »Bus-Error« (zwei Bomben) bestraft wird. Wir werden natürlich die Bomben vermeiden und auch auf geschützte Bereiche zugreifen, denn der Schutz besteht natürlich nur für Nicht-Assembler-Programmierer.

2.2 Fetch & Execute

Generell läuft ein Programm in einem Computer nach der Methode »Fetch and Execute«, wie die Amerikaner so schön prägnant sagen. Auf deutsch heißt das »Holen und Ausführen«. Die CPU holt sich aus dem Speicher einen Befehl und führt ihn aus. Danach holt sie sich automatisch den nächsten Befehl und führt diesen aus, usw., usw. Natürlich muß im Speicher etwas stehen, das die CPU holen und ausführen kann, und das nennt man dann Programm.

2.3 Programme sind nur Bytefolgen

Ein Programm ist nichts weiter als eine Folge von Bytes, die irgendwo im RAM oder ROM steht. Natürlich kann die CPU nicht wissen, wo das Programm im Speicher steht. Deshalb wird sie beim Start (Reset) per Hardware-Vorgabe sozusagen mit der Nase auf eine Anfangsposition gestoßen. Ab diesem Augenblick holt sich die CPU immer ein Wort (das sind 2 Byte nebeneinander, also 16 Bit) aus dem Speicher und dekodiert dieses Wort. Dabei kommt dann (hoffentlich) ein Befehl für die CPU heraus. Diesen Befehl arbeitet sie ab und holt dann das nächste Wort. Zu einem Befehl können Daten gehören. Beim Addier-Befehl zum Beispiel muß die CPU wissen, was addiert werden soll. Wieviel Datenworte zu einem Befehl gehören, ist auch im ersten Wort (dem Befehlswort) kodiert.

Der gesamte Speicher ist Byte für Byte von Null bis zum Ende durchnumeriert, diese Nummern der Speicherplätze nennt man Adressen. Die CPU arbeitet immer nur mit diesen Adressen und führt dazu intern einen Zähler, der immer auf die aktuelle Adresse zeigt, bei der sie gerade ist. Diesen Zähler nennt man »Program Counter«, kurz PC.

Hier ein Beispiel:

Adresse (PC)	Befehl	Daten
1000	Lösche	Wort
1004	Addiere	Operand 1, Operand 2
1010	Return	
1012		

Das »Listing« zeigt schematisch ein Programm, das bei Adresse 1000 beginnt. Um das Programm zu starten, muß man nur den PC auf 1000 setzen, und schon läuft es. Befehl 1 belegt die Adressen 1000 und 1001. Er hat in diesem Beispiel ein Datenwort auf Adresse 1002 und 1003. Die CPU arbeitet diesen Befehl ab und stellt dann den PC auf Adresse 1004. Zu Befehl 2 (auf 1004 und 1005) gehören 2 Datenworte (1006–1009), folglich muß Befehl 3 bei Adresse 1010 starten. Der 68000 kennt Befehle ohne Daten, die sind dann ein Wort lang, aber auch solche mit bis zu 4 Datenworten. D. h., beim 68000 kann ein einziger Befehl mit seinen Daten bis zu 10 Bytes (5 Worte) belegen. Wie Sie aus diesem Schema ersehen können, muß jeder Befehl auf einer geraden Adresse (Wortgrenze) beginnen, andernfalls passiert Übles.

Nun fragen Sie, wie das kommt? Ganz einfach: Sie können (und müssen) den PC verändern. Wenn nämlich ein Programm nicht nur einfach Befehl für Befehl abläuft, Sie also zum Beispiel ein GOTO benötigen, dann heißt das in Assembler zuerst einmal »GOTO Adresse«. Praktisch heißt das aber für die CPU »Setze PC = Adresse«. Wenn Sie da eine ungerade Adresse angeben, stürzt leider Ihr Programm ab.

In der Praxis tritt dieser Fehler auf, wenn Sie im Programm Daten definieren. Wenn Sie zum Beispiel den Text »Franz Meier« drucken wollen, müssen Sie irgendwo im Programm eine Byte-Folge mit den ASCII-Codes dieser Zeichen laden. Folgen dann hinter dem Text weitere Daten oder Programmtext, kommt es ganz auf die Länge des Textes an, ob diese auf eine gerade oder ungerade Adresse geraten. Für diesen Fall gibt es in vielen Assemblern einen Befehl wie EVEN, der Daten auf eine gerade Adresse justiert. Ist die Adresse schon gerade, hat EVEN keine weitere Wirkung. Ein EVEN zuviel schadet also nichts, eines zuwenig dagegen sehr.

2.4 Die Ausnahme ist die Regel

Der 68000 hat einen besonderen Mechanismus für die Behandlung sog. Exceptions (Ausnahmen). Da ein ST-Programm dies sehr häufig nutzt, man kann sagen, ohne Exceptions geht gar nichts, müssen wir uns damit intensiv befassen. Der 68000 kann durch externe Signale (zum Beispiel Reset) in den Ausnahmezustand gebracht werden, aber auch intern,

nämlich per Software. Der Auslöser kann ein Fehler sein, zum Beispiel Zugriff auf eine ungerade Adresse oder Absicht, nämlich ein Befehl. Insgesamt gibt es 256 Möglichkeiten, so eine Ausnahme auszulösen. Der 68000 hat für jede Exception eine Adresse fest vorgesehen, zum Beispiel gehört zu einem »Busfehler« die Adresse 8. Ab Adresse 8 steht in den nächsten 4 Byte wiederum eine Adresse, nämlich die der Routine, die den Fall Busfehler behandeln soll. So eine Adresse, in deren Bytes wiederum nur eine Adresse steht, nennt man Vektor (Zeiger). Die Vektor-Tabelle startet bei jedem 68000-Rechner bei Adresse 0 und belegt 256 x 4 = 1024 Byte. Was in den Vektoren steht, hängt vom Rechner ab. Die folgende Tabelle zeigt die ersten 12 Exception-Vektoren. Die Adressen sind hexadezimal notiert.

Vektor-Nr.	Adresse	Bedeutung
0	0000	Stack Pointer nach Reset
1	0004	Program Counter nach Reset
2	0008	Bus-Error ***
3	000C	Adreß-Error ***
4	0010	Illegaler Befehl ***
5	0014	Division durch 0
6	0018	vom CHK-Befehl ***
7	001C	vom TRAPV-Befehl ***
8	0020	Privileg-Verletzung ***
9	0024	TRACE ***
10	0028	Line-A-Emulator (Grafik)
11	002C	Line-F-Emulator ***

Bild 2.2: Der Anfang der Vektor-Tabelle

Für uns interessant sind dann noch die sog. Trap-Vektoren. Das sind die Vektoren 32 bis 47 (Adresse $0080 bis $00BC). Sie heißen Trap #0 bis Trap #15. Mit einem Trap-Befehl kann man ein Programm absichtlich in eine Falle (Trap) laufen lassen. Natürlich muß dann der Vektor in der zugehörigen Adresse auf eine sinnvolle Routine zeigen. Ich werde später auf einzelne Vektoren noch genau eingehen, vorerst nur soviel: Wenn Sie in Assembler schreiben »TRAP #1«, dann schaut der 68000 nach, welche Adresse im Vektor 33, also unter Adresse $0084 steht (das Dollar-Zeichen heißt hexadezimal) und lädt den PC mit dieser Adresse. Beim ST steht im Vektor 33 zum Beispiel (je nach TOS) $965E. Folglich wirkt der Befehl

```
TRAP #1
```

sinngemäß wie ein GOSUB $965E

2.5 Das hexadezimale Zahlensystem

Das hexadezimale Zahlensystem ist in Assembler üblich (und sehr vorteilhaft), machen Sie sich bitte ggf. mit diesem Zahlensystem vertraut. Hier ein Schnellkurs: Die Zahlenbasis ist nicht 10, wie im 10er-System, sondern 16. Für die nun fehlenden »Ziffern« von 10 bis 15 schreibt man A bis F. In dezimal sagt man für die Zahl 345 auch 5 Einer plus 4 Zehner plus 3 Hunderter. In »hex« ist die Basis 16, die Folge wäre also nicht 1, 10, 100, 1000, sondern 1, 16, 256, 4096. Sie wissen, »F« hat den Wert 15. Demnach ist FFFF = 15 * 4096 + 15 * 256 + 15 * 16 + 15 * 1 = 65535.

2.6 Ein Basic-Programm zum Üben der Hexerei

Bild 2.3 bringt ein kleines Programm in GfA-Basic zum Üben.

```
While 1
      Input "Eine Zahl n ($n wenn hex) ";A$
      If Left$(A$,1)<>"$" Then
           Print Hex$(A$,Len(A$)-1)
      Else
          A$=Right$(A$,Len(A$)-1)
             L=Len(A$)
          X%=0
          For I=L To 1 Step -1
             X%=X%+Val("&h"+Mid$(A$,I,1))*16^(L-I)
          Next I
          Print X%
      Endif
Wend
```

Bild 2.3 Hex-Dezi-Konvertierung in Basic

Wenn Sie eine Dezimalzahl eingeben, dann gibt sie das Programm in hex aus. Umgekehrt, geben Sie eine Hex-Zahl ein (erkenntlich am $ als erstes Zeichen), erhalten Sie deren Wert in dezimal.

2.7 Das duale Zahlensystem

Sozusagen noch eine Stufe tiefer (noch näher am Computer) ist das duale Zahlensystem. Hier ist die Basis 2, womit in diesem System nur die Ziffern 0 und 1 erlaubt sind. Am

einfachsten kann man eine Dualzahl in dezimal umrechnen, indem man sich die Wertigkeit darüber schreibt.

Hier ein Beispiel:

Dezimale Wertigkeit	32	16	8	4	2	1
Dualzahl	1	0	1	1	0	1

Das Ergebnis wäre dann 32+8+4+1 = 45

Die Verbindung zum hexadezimalen Zahlensystem ist recht einfach zu erledigen. Nehmen wir an, wir hätten diese Dualzahl

1010 0101

Sie sehen schon, ich habe sie in Vierergruppen geteilt. Lege ich wieder die Wertigkeit darüber, sieht das so aus:

```
8 4 2 1     8 4 2 1
-----------  -----------
1 0 1 0     0 1 0 1
```

Das ergibt (von links) dezimal 10 und 5. In »hex« schreibt man für 10 aber A, also hieße die Zahl in »hex« A5. Auch hier wieder mit Bild 2.4 ein GfA-Basic-Programm zum Üben von Dualzahlen, auch Binärzahlen genannt:

```
While 1
        Input "Eine Zahl n (%n wenn binär) ";A$
        If Left$(A$,1)<>"%" Then
            Print Bin$(A$,Len(A$)-1)
        Else
           A$=Right$(A$,Len(A$)-1)
           L=Len(A$)
           X%=0
           For I=L To 1 Step -1
              X%=X%+Val("&h"+Mid$(A$,I,1))*2^(L-I)
           Next I
           Print X%
        Endif
Wend
```

Bild 2.4 Umrechnung binär in dezimal und zurück

Wenn Sie eine Dezimalzahl eingeben, dann gibt sie das Programm in binär aus. Umgekehrt, geben Sie eine Binär-Zahl ein (erkenntlich am % als erstes Zeichen), erhalten Sie deren Wert in dezimal. Das Zeichen Prozent (%) ist in Assembler der Präfix für Dualzahlen.

2.8 Der Sinn der Bomben und Pilze

Sie haben sicherlich beim ST schon einige Systemabstürze erlebt und dann einige Bomben (im alten TOS Pilze) auf dem Schirm gesehen. Die in Bild 2.2 mit *** gekennzeichneten Vektoren sind vom ST mit »Bombendrohung« belegt. Vektor 5 (Division durch Null) reagiert nicht, weil sein Inhalt auf eine Routine zeigt, in der nur Return steht. Die »Bomben-Vektoren« zeigen alle auf dieselbe Routine. Dieses Unterprogramm stellt die Vektor-Nummer als entsprechende Anzahl Bomben auf dem Schirm dar. Drei Bomben zum Beispiel heißt dann Adreßfehler. Sinn der Übung ist es, auch dann eine Warnmeldung ausgeben zu können, wenn beim Absturz der Zeichensatz zerstört ist oder (Fehler beim Start) noch gar nicht initialisiert war.

2.9 Stack: Funktion und Aufgaben

Das kürzeste Programm, das Sie für den ST schreiben können, heißt in Assembler:

```
CLR.W   -(SP)
TRAP    #1
```

und schon haben Sie den Stack benutzt. SP (oder A7, was dasselbe ist) werden Sie am häufigsten in jedem Programm finden, ein Grund, uns auch das Ding genauer anzusehen. Der Stack ist ein Speicher (ein Stück RAM) mit besonderen Eigenschaften. Man nennt ihn auch LIFO für »Last In, First Out« oder Stapelspeicher. Sie packen Daten auf den Stack, indem Sie etwas auf den Stapel tun. Sie können immer nur von oben (vom »Top of Stack«) etwas wegnehmen. D. h., wenn Sie die Daten A, B und C in dieser Reihenfolge auf den Stack packen, können Sie sie nur in der Folge C, B, A zurücklesen. Der Trick ist nun, daß die CPU tatsächlich niemals die Daten vom Stack nimmt, sondern nur einen Zeiger auf die Daten verändert. Gesteuert wird dieses durch den sog. Stapelzeiger, neudeutsch Stackpointer oder kurz SP. Noch eine Vorbemerkung: Der Stack wächst von oben (den hohen Adressen) nach unten (zu den niedrigen Adressen hin). Die Anweisung »Packe A auf den Stack« bewirkt 2 Schritte:

1. Erniedrige SP
2. Kopiere A in den Speicherbereich, auf den SP nun zeigt.

Die Umkehr, nämlich »hole A vom Stack«, hat zur Folge:

1. Kopiere Daten, auf die SP zeigt, nach A
2. Erhöhe SP

Was das u.a. für einen Sinn macht, soll Bild 2.5 zeigen. Es geht um Unterprogramme, hier am Beispiel von Basic. Sie können sich aber auch die Zeilennummern als Adressen vorstellen. Rechts ist immer ein Stück des Stacks, daneben der Stackpointer gezeichnet. Der Befehl »GOSUB 100« bewirkt dreierlei.

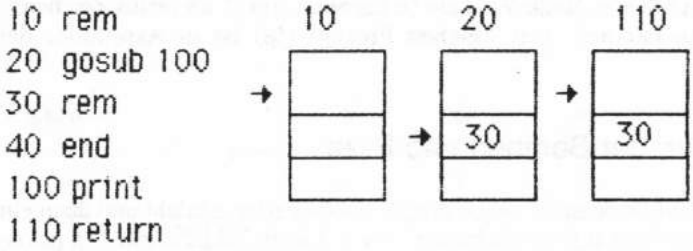


Bild 2.5 Sinn des Stacks am Beispiel Basic

1. Erniedrige SP
2. Packe die nächste Zeilennummer (hier 30) auf den Stack (auf die Speicherstelle, auf die jetzt SP zeigt)
3. Springe zur Zeile 100

Das Return in Zeile 110 hat zur Folge:
1. Hole Zeilennummer, auf die SP zeigt
2. Erhöhe SP
3. Springe zur Zeile 30.

Nun fragen Sie vielleicht, warum SP vom GOSUB erniedrigt und vom RETURN erhöht wird? Nun, schauen Sie auf Bild 2.6. Hier ruft das Unterprogramm ein weiteres Unterprogramm auf. Nun stehen nach Zeile 110 zwei Zeilennummern (genau Return-Adressen) auf dem Stack. Das Return von Zeile 210 stellt den SP auf Zeile 30 und springt dann zu 120, das Return in Zeile 120 stellt den SP wieder zurück und springt dann zu Zeile 30. Der SP steht wieder auf seinem Ausgangswert, »we are home «.

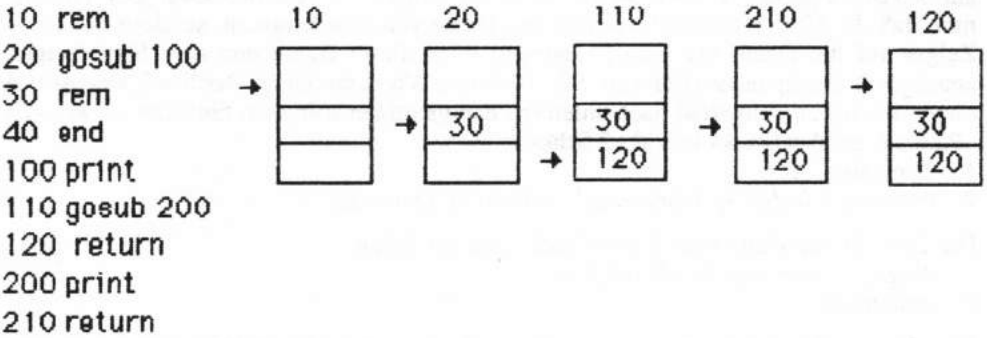


Bild 2.6: Stack im Fall »Unterprogramm ruft Unterprogramm«

Langer Rede, kurzer Sinn: mit dem Stackmechanismus können Unterprogramme beliebig tief geschachelt werden. Jedes Return erhöht den SP wieder und so hangelt man sich dann zurück. Aber Achtung, was passiert hier?

```
10 GOSUB 20
20 GOSUB 10
```

Da jedes GOSUB den SP erniedrigt, aber das Gegenstück, nämlich das RETURN, fehlt, wächst der Stack nach unten. Er wird dann recht bald in Ihren Programm-Code laufen und den mit Return-Adressen überschreiben. Ergebnis: Totaler Crash, auch in Basic. Probieren Sie es einmal.

Die zweite Anwendung für den Stack ist die Parameterübergabe an Unterprogramme. Prinzipiell läuft das so: Es gibt in Assembler (nicht in Basic) die Befehle »Packe Daten auf den Stack« und »hole Daten vom Stack« (Sie wissen, jeder Befehl impliziert ein Verändern des Stack-Pointers). Da kann ich dann sinngemäß schreiben:

```
10 A auf den Stack
20 B auf den Stack
30 GOSUB 100 (Return-Adresse auf Stack).
```

und dann im Unterprogramm:

```
100 Hole Return-Adresse vom Stack (und merke sie)
110 Hole B vom Stack
120 Hole A vom Stack
130 Rechne mit A und B
140 Springe zur Return-Adresse
```

Was aber, wenn Ihr Unterprogramm mit Return enden soll? Dann schreibt man:
1. Return-Adresse auf den Stack
2. Daten auf den Stack
3. GOTO Unterprogramm

Im Unterprogramm:
1. Daten vom Stack
2. mit Daten arbeiten
3. RETURN

Beim ST werden Sie diesen Mechanismus sehr häufig antreffen:
1. Daten auf den Stack
2. TRAP #x (heißt praktisch GOSUB).

Alle Routinen des TOS werden auf diese Art aufgerufen, und diese TOS-Routinen werden wir in jedem Programm nutzen. Um es gleich zu sagen: Es gibt in Assembler keinen PRINT-Befehl, sondern nur die Möglichkeit, Bytes in einen Speicherbereich zu schreiben, der (vom Video-Kontroller) auf dem Bildschirm abgebildet wird. Überhaupt heißt Assembler-Programmierung primär, Daten von einer Adresse auf eine andere Adresse zu bewegen. Auch die Peripherie-Geräte (Tastatur, Floppy usw.) liegen beim ST innerhalb des Adreßbereichs (man nennt das »memory mapped«). Die Geräte werden angesprochen, indem man bestimmte Daten in diese Adressen schreibt oder von ihnen liest.

Sie sehen also schon, die Adressierung als solche ist ganz wesentlich. Man kann eine Adresse auf sehr viele unterschiedliche Arten ansprechen, ein Beispiel hatten wir schon mit dem SP. Ich kann da sagen »stelle den SP auf die Adresse 4711«. Ich kann aber auch sagen »hole Daten von der Adresse, auf die SP gerade zeigt« (ohne zu wissen, wohin er zeigt).

Das waren schon 2 Adressierungsarten. Insgesamt kennt der 68000 aber 12, und mit diesen 12 Adressierungsarten werden wir uns im nächsten Kapitel beschäftigen. Sie sind sozusagen der Schlüssel zum 68000.

Kapitel 3

Adressen, Daten und Befehle

Register

Adressierungsarten

Datentypen

Struktur der Befehle des 68000

Bisher hatten wir gelernt, daß Daten im RAM oder ROM stehen. Daneben gibt es aber einen ganz speziellen RAM, der ein Teil der CPU ist. Dieser Speicherbereich besteht aus Gruppen von je 32 Bit und jede dieser Gruppen nennt man ein Register.

3.1 Tempo durch Register

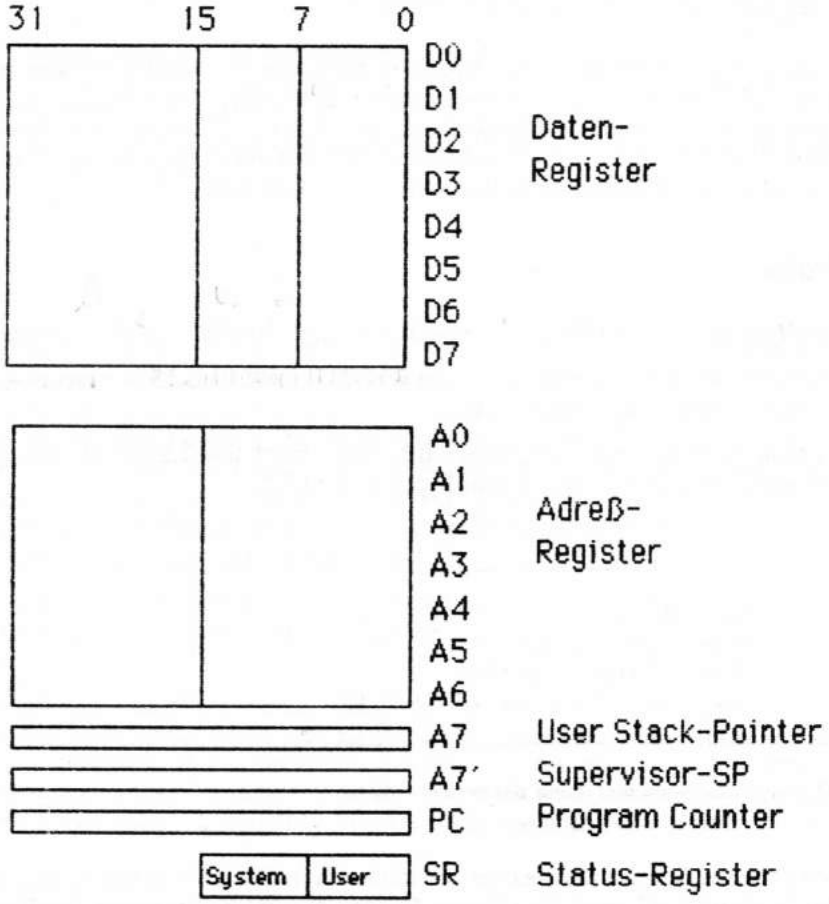


Bild 3.1: Das Register-Modell des 68000

Die Register werden nicht über Adressen, sondern über Namen angesprochen. Der Vorteil von Registern im Vergleich zum übrigen Speicher ist, daß sich die Register auf demselben Chip wie die CPU befinden, und die CPU den Zugriff auf diese Register durch spezielle Befehle unterstützt.

Natürlich entfällt auch der Umweg über die MMU und den Bus. Damit sind Registeroperationen wesentlich schneller als Zugriffe auf den Hauptspeicher und bieten (wegen der speziellen Befehle) einiges mehr an Komfort.

3.2 Das Register-Modell des 68000

So gesehen, ist eine CPU mit vielen Registern besser als eine solche mit wenigen. Der 68000 hat viele Register, nämlich: 8 Datenregister, 7 Adreßregister, 2 Stackpointer, einen Programmzähler (PC) sowie ein Statusregister. Bild 3.1 zeigt den kompletten Registersatz des 68000. Wie Sie sehen, sind die Register D0-D7 und A0-A7 plus PC je 32 Bit breit, das sind vier Bytes. Im Bild sind die Bits von 0 bis 31 numeriert.

3.3 Datentypen

In einem Byte zählt man die Bits von 0 (niedrigstwertige Bits) bis 7 (höchstwertige Bits).

Zwei Byte (16 Bit) nennt man Wort. Dessen Bits zählen von 0 bis 15.
Zwei Worte (32 Bit) sind ein Langwort.

Man spricht auch von den Datentypen Bit, Byte, Wort und Langwort. Die größten darstellbaren Zahlen in Abhängkeit vom Typ zeigt Bild 3.2.

Bit = 2↑1-1 = 1
Byte = 2↑8-1 = 255
Wort = 2↑16-1 = 65535
Langwort = 2↑32-1 = 4,294,967,299

Bild 3.2: Die Datentypen und damit darstellbare Werte

Nun verstehen Sie auch die senkrechten Trennlinien in Bild 3.1. In den Datenregistern kann man Bytes, Worte und Langworte ablegen. Bei den Adreßregistern ist der Typ Byte nicht möglich. Die Stackpointer (A7) sind immer »long«.

Außerdem gibt es noch den Type BCD (Binary Coded Decimal). In diesem Falle wird ein Byte in 2 Halbbytes (nennt man Nibble) geteilt. Mit den 4 Bits eines Nibbles kann man nun die Zahlen 0–15 darstellen. Auf 10 bis 15 wird dabei aber verzichtet, gültig sind in der BCD-Darstellung nur die Werte von 0 bis 9. Damit kann man in einem »BCD-Byte«

immer 2 Zehner-Stellen darstellen. Ein Wort reicht also für eine 4stellige Dezimalzahl. Braucht man mehr Stellen, muß man die entsprechende Anzahl Bytes sozusagen nebeneinanderlegen. Nun gibt es eine Unmenge kluger Algorithmen zum Thema BCD-Rechnerei. Für die armen Leute ohne 68000 sind die sehr wichtig. Wir können darauf verzichten, weil im 68000 die passenden Befehle dafür schon eingebaut sind.

Typ-Angabe ist die erste Bürgerpflicht

Solange Sie nur mit Registern arbeiten, spielt die Größe (fast) keine Rolle, wenn Sie aber ein mit allen 32 Bit gefülltes Register in den RAM kopieren, belegt es da 4 Bytes. Das ist ziemlich unpraktisch, weil Sie sehr oft mit Bytes oder Worten auskommen könnten, also auch mit weniger Speicher. Deshalb gibt es beim 68000 die Möglichkeit – genau: die Pflicht, bei jeder Operation, die Daten bewegt, anzugeben, welcher Typ dabei gilt. Ein Beispiel: Der Datentransfer geschieht mit dem Befehl MOVE und zwar mit der Syntax »MOVE Quelle,Ziel«. Tatsächlich bewegt der Befehl MOVE die Daten nicht, sondern kopiert sie, er kopiert von der Quelle auf das Ziel. Um die Daten im Register D3 auf die Adresse 4711 zu kopieren, schreibt man

```
MOVE.B D3,4711 oder
MOVE.W D3,4711 oder
MOVE.L D3,4711
```

Im Falle .B wird ein Byte kopiert, also die Bit 0–7 von D3, im Falle .W ist es ein Wort (Bit 0–15) bzw. im Falle .L das ganze Register. Im Hauptspeicher (hier ab Adresse 4711) werden dann entsprechend 1, 2 oder 4 Bytes belegt. In welcher Reihenfolge dabei die verschiedenen Datentypen im RAM stehen, sollten Sie wissen, nämlich so, wie man sich das denkt. Steht beispielsweise in D0 das Wort $AABB und wird D0 mit einem Move.W-Befehl auf die Adresse 1000 kopiert, so steht $AA in 1000 und $BB in 1001. Nur den Kollegen aus der 8-Bit-Ecke und denen, die vom IBM-PC kommen, sei noch einmal deutlich gesagt: Der 68000 speichert Daten in der richtigen Reihenfolge und nicht wie die »8-Bitter/8088er«, die Bytes eines Wortes vertauschen!

Wegen dieser freien Auswahl des Datentyps haben Sie leider auch die Pflicht, ihn bei den meisten Befehlen anzugeben. Fehlt der Typ, nehmen die meisten Assembler den Typ »Wort« an.

Um auf die Register zurückzukommen: Der Hauptunterschied zwischen Daten- und Adreßregistern ist, daß bei letzteren die Typen Bit und Byte nicht erlaubt sind. Ansonsten können Sie durchaus auch Daten in Adreßregistern speichern und Adressen in Datenregistern, nur guter Programmierstil ist das nicht. Das Statusregister hat einen ganz besonderen Zweck. Damit wird in Assembler IF-THEN realisiert, ich komme noch (sehr ausführlich) darauf zurück.

3.4 Befehle

Wieviele Assemblerbefehle es gibt, ist gar nicht so einfach zu sagen. Das liegt daran, daß ein Befehl je nach Adressierungsart (und weiteren Varianten) ganz unterschiedliche Wirkungen zeigt. Beginnen wir mit dem Befehlsaufbau. Ein Befehl kann haben: keinen Operanden oder einen oder zwei. Die Operanden können im Befehlswort selbst enthalten

sein oder belegen bis zu vier weitere Worte, die dem Befehlswort unmittelbar folgen. Darüber brauchen Sie sich aber vorerst wenig Sorgen zu machen. Im Quelltext schreiben Sie den Befehl und die Operanden einfach hin, wieviele Worte das dann werden, ist Sache des Assemblers.

Ein Beispiel für einen Befehl mit keinem Operanden ist RTS (Return from Subroutine), was dem RETURN in Basic entspricht. Einen Operanden hätte der Befehl »CLR D0«. Das heißt Clear (Lösche [fülle mit Nullbits] den Operanden D0 [das Register D0]). Ein Beispiel für einen Befehl mit 2 Operanden:

```
MOVE.L A3,A4
```

Damit wird das Langwort im Register A3 nach A4 kopiert.

3.5 Sinn und Zweck der Adressierungsarten

Eine CPU ist um so besser, je mehr sinnvolle Adressierungsarten sie hat, und hier glänzt der 68000 ganz besonders. Dieser Luxus macht die Sache etwas schwierig, denn das alles will gelernt sein, und hier liegt auch die Barriere für die Kollegen, die es gewohnt sind, mit den wenigen (und primitiven) Adressierungsarten der »8-Bitter« auszukommen. Andererseits, wenn Sie das Thema beherrschen, dann beherrschen Sie auch den 68000.

Um zu zeigen, worum es geht: Im Beispiel von eben

```
MOVE.L A3,A4
```

wurde der Inhalt des Registers A3 nach A4 kopiert. Schreibe ich hingegen

```
MOVE.L (A3),(A4)
```

heißt das, daß die Inhalte der Register als Adressen zu sehen sind. Hat z. B. im Moment des Befehls A3 den Wert 4711 und A4 ist gleich 5711, dann wird ein Langwort von Adresse 4711 (da startend und Byte für Byte) nach Adresse 5711 kopiert.

Wir haben nun schon 2 Adressierungsarten kennengelernt, nämlich »Register direkt« (MOVE.L A3,A4) und »Register indirekt« (MOVE.L (A3),(A4). Um noch eine Stufe höher zu gehen, schauen wir uns »Adreßregister indirekt mit Postinkrement« an. Das sieht z. B. so aus:

```
MOVE.W (A0)+,D0
```

Im Klartext: Kopiere das Wort, auf das A0 zeigt nach D0 und erhöhe danach (post) A0 um 2. Zwei deshalb, weil ein Wort 2 Bytes hat. Bitte merken: Der 68000 ist eine Byte-Maschine, jede Adresse zeigt auf 1 Byte. »MOVE. (A0)+,D0« würde ein Langwort kopieren und danach A0 um 4 inkrementieren. Die nächste Variante wäre »Adreßregister indirekt mit Predekrement. Ein Beispiel:

```
MOVE.L D0,-(A5)
```

In diesem Falle wird vorab (pre) 4 (Langwort hat 4 Bytes) von A5 subtrahiert, dann wird D0 dahin kopiert, wohin A5 nun zeigt. Das hatten wir doch schon mal? Sie erinnern sich an den Stack aus Kapitel 2! Daten werden auf den Stack gebracht, indem man den

Stackpointer (SP) erniedrigt und dann die Daten auf die Adresse kopiert, auf die SP zeigt. Daten werden vom Stack geholt, indem man sie von der Adresse holt, auf die SP zeigt und dann SP erhöht. Das wäre dann unser schon bekanntes

```
MOVE.L (A5)+,D0
```

Tatsächlich kann man so jedes Adreßregister als Stackpointer einsetzen. Die Besonderheit des Registers A7, das auch in vielen Assemblern SP heißt, liegt darin, daß dieses durch Befehle wie JSR (Jump to Subroutine) und RTS (Return) angesprochen wird. Gleiches kann man aber ebensogut mit anderen Registern erledigen, z. B. kann man anstatt RTS auch schreiben

```
MOVE.L (A7)+,A0
JMP (A0)
```

Der MOVE-Befehl holt die Return-Adresse vom Stack in das Register A0, danach erfolgt ein Sprung (Jump) zur Adresse, auf die A0 nun zeigt. Sie sagen, warum der Umstand, ein RTS ist doch viel einfacher! Recht haben Sie, aber trotzdem werden Sie diese Lösung in Programmen sehen, die z. B. von Basic aus aufgerufen werden und zwar so:

```
MOVE.L (A7)+, 4711
```

Viele andere Befehle:

```
MOVE.L 4711,A0
JMP (A0)
```

Mit dem ersten MOVE-Befehl wird die Return-Adresse auf einen sicheren Platz in den RAM geholt (4711 ist hier nur symbolisch gemeint). Man sagt auch, die Return-Adresse wird gerettet. Wenn irgend etwas schiefgeht, kann ich dann immer noch mit Hilfe dieser Adresse zu Basic zurück, egal wo der Stackpointer gerade steht. Nach diesem Ausflug in die Praxis, der einmal andeuten sollte, wofür man verschiedenartige Adressierungsarten braucht, wieder zurück zur Theorie. Bild 3.4 zeigt eine Liste aller Adressierungsarten und noch etwas mehr.

Adressierungsart	Kürzel	Modus	Register
Datenregister direkt	Dn	000	Dn
Adreßregister direkt	An	001	An
Adreßregister indirekt (ARI)	(An)	010	An
ARI mit Postinkrement	(An)+	011	An
ARI mit Predekrement	-(An)	100	An
ARI mit Adreßdistanz	d16(An)	101	An
wie vor plus Index	d8(An,Rn)	110	An
Absolut kurz	$XXXX	111	000
Absolut lang	$XXXXXXXX	111	001
PC-Realtiv mit Adr.-Distanz	d16(PC)	111	010
PC-Realtiv mit Adr.-Distanz plus Index	d8(PC,Rn)	111	011
Konstante, Statusregister	#, SR,CCR	111	100

Bild 3.4: Liste aller Adressierungsarten

Zuerst notieren Sie bitte nur, daß eine Adresse aus mehreren Angaben zusammengesetzt sein kann. Die CPU errechnet daraus die endgültige Adresse, auch effektive Adresse (ea) genannt. Wie Sie schon wissen, belegt das Befehlswort 16 Bit. Die 4 höchstwertigen davon beschreiben den Befehl an sich. Die übrigen 12 teilen sich in 2 Gruppen von 6 Bit, die die Adressierungsart von Ziel und Quelle (so vorhanden) angeben. Die 6 Bit je Operand wiederum teilen sich in 2 Gruppen von 3 Bit, die eine Gruppe heißt Modus, die zweite Register. Mit 3 Bit sind die Zahlen 0 bis 7 darstellbar, deshalb gibt es auch die Register A0–A7 bzw. D0–D7. Es gibt aber mehr als 7 Adressierungsarten, was damit erreicht wird, daß nicht bei jedem Adressier-Modus alle Register erlaubt sind. Überhaupt, und das ist wichtig zu wissen, sind bestimmte Adressierungsarten nicht für den Quell- und (gleichzeitig) den Zieloperanden erlaubt und außerdem auch nicht für jeden Befehl. Zu diesem Thema finden Sie mehr Informationen im Anhang. Im Bild 3.4 sind Modus und Register als Binärzahlen dargestellt. Steht dort »An« oder »Dn«, können Sie dafür %000 bis %111 (dezimal 0 bis 7) einsetzen.

3.6 Die Adressierungsarten im Detail

Wie Sie nun ganz richtig erkannt haben, geht aus dem Befehlswort und den darin kodierten Adressierungsarten auch hervor, wieviele Worte der Befehl im Speicher belegt. Register-Register-Adressierung (z. B. MOVE A0,A1) kommt mit einem Wort aus, geben Sie hingegen eine absolute Adresse an, kommt mindestens noch ein Wort hinzu. D. h. in den 16 Bit des Befehlswortes stecken alle Informationen, die die CPU braucht, um den Befehl zu dekodieren. So ähnlich arbeiten auch sog. Disassembler, das sind Programme, die aus dem Maschinencode wieder den Klartext der Assemblersprache bilden. Solange Sie aber ein

solches Programm nicht schreiben wollen, können (und sollten) Ihnen diese Bitmuster herzlich egal sein. Es gibt viele Hacker, die den Hex-Code (die Maschinensprache) der »8-Bitter« lesen können, wie andere Leute die Zeitung. Diese Übung ist beim 68000 aussichtslos, also sehen wir die Sache von der praktischen Anwendung her, nämlich alle Adressierungsarten an je einem Beispiel.

3.6.1 Register direkt

Eines der Register wird direkt angesprochen.

Beispiel:

```
CLR   D0   (Lösche D0).
```

3.6.2 Adreßregister indirekt (ARI)

Der Inhalt des Registers ist eine Adresse, auf diese wirkt die Operation.

Beispiel:

```
MOVE  (A0),D0.  ARI bitte merken!
```

Das Wort, dessen Adresse in A0 steht, wird nach D0 kopiert.

3.6.3 ARI mit Postinkrement

Wirkt wie ARI, nur wird anschließend das Register inkrementiert. Beispiele:

```
MOVE.B    A(0)+,D0.    Kopie, dann A0=A0+1
MOVE.W    A(0)+,D0.    Kopie, dann A0=A0+2
MOVE.L    A(0)+,D0.    Kopie, dann A0=A0+4
```

3.6.4 ARI mit Predekrement

Wie vor, nur wird das Register vor der Operation erniedrigt.

Beispiel:

```
MOVE  -(A0), D0   A0=A0-2,   dann Kopie
```

3.6.5 ARI mit Adreßdistanz

Die effektive Adresse ist die Summe von Inhalt des Registers plus Adreßdistanz. Die Adreßdistanz ist eine vorzeichenbehaftete 16-Bit-Zahl im Bereich -32668..32767.

Beispiel:

```
MOVE  -100(A0), D0
```

Wäre A0=500, würde das Wort von Adresse 400 nach D0 kopiert

3.6.6 ARI mit Adreßdistanz und Index

Nun wird es kompliziert. Zuerst: Die Adreßdistanz ist jetzt nur noch eine vorzeichenbehaftete 8-Bit-Zahl im Bereich von -128..127. Nun darf aber noch ein weiteres Register angeben werden.

Ein Beispiel:

```
MOVE  100(A0,D0),4711
```

100 ist die Adreßdistanz, A0 enthält die Basisadresse, in D0 steht der Index. Alle drei werden addiert. Die Summe ist eine Adresse; das Wort (Byte, Langwort), das da steht, wird ins Ziel (hier Adresse 4711) kopiert. Beim Index darf auch ein anderer Typ angegeben werden, also auch D0.B oder D0.L wären erlaubt, bei Adreßregistern als Index natürlich nur An.W und An.L (0<n<7). Auch der Index ist vorzeichenbehaftet, womit er im Falle Langwort im Bereich von 2 Giga-Byte liegen muß (wenn wir die mal hätten). Dieser Befehl ist ideal für die Abarbeitung von Tabellen und Arrays. Oft wird dabei die Adreßdistanz nicht benötigt (die Laufvariable steht im Indexregister), weshalb man oft die Form von z. B. »0(A3,D4.L)« sieht.

3.6.7 Absolute Adressierung

Dies ist der einfachste Fall.

Beispiel:

```
MOVE  4711,5713
```

Das Wort von Adresse 4711/12 wird auf Adresse 5713/14 kopiert. Die CPU unterscheidet dabei noch zwischen kurz und lang (Adreßbereich nur 64 K oder die vollen 16 Megabyte des 68000). Praktisch merken Sie den Unterschied kaum (die lange Adresse erfordert mehr Bytes in der Befehlslänge und ist etwas langsamer).

3.6.8 Konstanten-Adressierung

Auch wieder etwas ganz einfaches. Um eine Konstante zu bewegen, brauchen Sie nur das Zeichen # davor zu setzen.

Um z. B. das ASCII-Zeichen A in das Register D0 zu laden, schreiben Sie

```
        MOVE  #65,D0
oder    MOVE  #'A',D0
```

3.6.9 PC-relative Adressierung

Da muß ich etwas ausholen, damit Sie dieses Feature auch würdigen können. Sobald Sie in einem Assembler-Programm eine absolute Adresse angeben, ist das Programm an einen Ort im Speicher gebunden. Auch z. B. »(A0)« (indirekt) ist in diesem Sinne absolut, denn vorher mußten Sie ja A0 mit einer Adresse versorgen.

Der von Ihnen geplante Adreßbereich kann aber schon belegt sein, also muß Ihr Programm auch an einem anderen Ort laufen können. Dazu gibt es zwei Möglichkeiten. Erstens, das Programm ist verschiebbar (relokatibel). Dafür sorgt der Assembler, indem er mit dem Programm eine Tabelle aller absoluten Adressen abspeichert. Der Lader (oder das Programm selbst oder eine Utility) kann dann diese Adressen korrigieren, indem sie die Differenz zwischen geplanter und tatsächlicher Start-Adresse auf alle absoluten Adressen lt. Tabelle addieren. Die zweite Möglichkeit ist, das Programm lageunabhängig (Position Independend) zu schreiben. In einem solchen Programm dürfen dann eben keine absoluten Adressen vorkommen, und genau da hilft der 68000 mit der PC-relativen Adressierung. Dabei wird die Adresse gerechnet als aktueller Stand des PC + Offset. Offset ist auch hier wieder auf -32768..32767 begrenzt.

Beispiel: `MOVE 100(PC),D0`

PC-relativ mit Adreßdistanz und Index

Hier gilt sinngemäß das für »ARI mit Adreßdistanz und Index« Gesagte, nur daß die Basis-Adresse hier PC+2 ist.

Beispiel: `MOVE 100(PC,A0.W),D0`

Das wär's vorerst an Theorie. Es fehlt zwar noch allerlei, aber das wird an passender Stelle anhand praktischer Beispiele erläutert. Im nächsten Kapitel kommen die ersten Listings. Außerdem müssen wir uns um die Bedienung von Editor, Assembler, Linker und Batch-Prozessor kümmern. Zum TOS wäre auch noch einiges zu sagen, schließlich wollen wir die Räder nicht neu erfinden, sondern alles, was da schon eingebaut ist, kräftig nutzen.

Kapitel 4

Ganz schnell zur Praxis

Hier geht es um das TOS,
die Bedienung des Assemblers
und natürlich die ersten Listings.

4.1 Ein Schnellkurs in Sachen TOS

Das Betriebssystem des ST heißt nach dem Boß von Atari »Tramiel Operating System«, kurz TOS. Die Aufgabe eines jeden OS (Operating System) ist es, die Verbindung des Computers mit der Außenwelt herzustellen. Dinge wie Zeichen von der Tastatur lesen, Zeichen auf dem Bildschirm darstellen oder Dateien von einer Diskette lesen sind typische Aufgaben des TOS.

Aus Sicht des Programmierers ist das TOS eine Sammlung von Routinen (Unterprogrammen), die er alle benutzen darf. Einige davon, wie zum Beispiel Laden und Starten eines Anwenderprogramms, sind dem »Normalverbraucher« zugänglich, alle nur dem Assembler-Programmierer. Jedes dieser Unterprogramme startet natürlich bei einer bestimmten Adresse, und demnach könnte man so ein Programm in der Form »JSR Adresse« aufrufen. Praktisch tut man das nicht, denn dann würde jede Änderung im OS dazu führen, daß sich einige oder alle dieser Adressen verschieben und somit alle »alten« Programme nur noch Makulatur wären.

4.2. Aufruf von TOS-Routinen

Um also von absoluten Adressen unabhängig zu sein, arbeiten alle guten OS nach diesem Schema: Alle Unterprogramme erhalten eine Nummer, Funktionsnummer genannt. Im OS steht eine Tabelle, in der notiert ist, welche Adresse zu jeder Funktionsnummer gehört. Das OS hat nun eine Routine, deren Adresse sich nie ändert. Das ist der Dispatcher. Um ein Unterprogramm aufzurufen, übergibt man dem Dispatcher die Funktionsnummer. Dieser berechnet danach (und mit Hilfe der Tabelle) die Adresse der Routine und ruft sie auf.

Nun aber zum ST: Hier wird ein Dispatcher (tatsächlich hat der ST deren 3) über den Trap #1 gerufen. Sie können den Trap #1 wie ein JSR (Jump to Sub Routine) auffassen, sprich, danach geht es mit dem Folgebefehl weiter. Die Parameterübergabe geschieht immer nach diesem Muster:

1. Parameter auf den Stack
2. Funktionsnummer auf den Stack
3. Trap #1
4. Stackpointer in den Zustand wie vor dem Aufruf bringen.

Manche Routinen haben keine Parameter, dann entfüllt Schritt 1. Es gibt nur eine Ausnahme, bei der auch Schritt 4 entfüllt, das ist die Routine TERM (Terminate = beenden), weil dort der Trap-Befehl die Rückkehr zum aufrufenden Programm (meistens dem Desk-top) veranlaßt.

4.3 Form eines Assemblerprogramms

Jedes Assemblerprogramm besteht aus den Feldern Marke, Befehle, Operanden (falls vorhanden) und Kommentar. Hier ein Muster:

Marke	Befehl	Operand(en)	Kommentar
Start	clr	d0	; Lösche Register
	move	d0,d1	; Befehl mit 2 Operanden
weiter			; nur eine Marke in der Zeile
	rts		; kein Operand

Der Kommentar muß nicht sein, er trägt aber zur Lesbarkeit bei. Je nach Assembler muß er mit einem Semikolon oder Stern beginnen, bei manchen Assemblern reicht auch die Position im Kommentarfeld aus, damit der Assembler erkennt, daß es sich um Kommentar handelt. Steht der Kommentar allein in einer Zeile, muß »;« oder »*« sein.

Die Marke (Label) wird nur in einigen Fällen gebraucht. Sie kann auch allein in einer Zeile stehen, sie wirkt aber immer auf die nächste Zeile mit einem Befehl. In manchen Assemblern muß der Marke ein Doppelpunkt folgen, dann aber nur, wenn sie im Markenfeld steht, nicht wenn sie angesprochen wird.

Ausgenommen die Sonderfälle »nur Marke« oder »nur Kommentar« muß ein Befehl in einer Zeile stehen, und, so vorhanden, auch dessen Operand(en). Die einzelnen Felder müssen durch mindestens eine Leerstelle voneinander getrennt sein. Meistens benutzt man die Tabulator-Taste (8ter Abstand). Nehmen Sie aber einen Texteditor, der dafür Blanks erzeugt.

4.4 Das erste Listing: Ausgabe von Zeichen

Nun zu unserem ersten Programm lt. Bild 4.1

```
* A1 mein erstes Programm

        move    #'h',-(sp)   ; das 'h' soll gedruckt werden
        move    #2,-(sp)     ; Funktion CONOUT
        trap    #1           ; GEMDOS aufrufen
        addq.l  #4,sp        ; Stack Pointer auf Ausgangswert

        move    #'a',-(sp)   ; das 'a' soll gedruckt werden
        move    #2,-(sp)     ; Funktion CONOUT
        trap    #1
        addq.l  #4,sp        ; Stack Pointer auf Ausgangswert

; nun machen Sie weiter mit l l o
```

```
; jetzt warten wir auf einen Tastendruck
        move    #1,-(sp)      ; Funktion CONIN
        trap    #1            ; GEMDOS aufrufen
        addq.l  #2,sp         ; Stack korrigieren
; nun beenden wir das Programm
        move    #0,-(sp)      ; Funktion TERM(inate)
        trap    #1
        end                   ; Anweisung an den Assembler
```

Bild 4.1: Ausgabe von Zeichen

Das Programm soll »hallo« auf den Schirm schreiben, auf einen Tastendruck warten und dann zum Schreibtisch zurückkehren. Drei Befehle müssen wir hierzu kennen:

»move« oder »move.w« mit der Syntax »move Quelle,Ziel« schreibt oder kopiert (also moves) ein Wort. Wenn man eine Konstante bewegen will (die dann nur Quelle sein kann), muß man das Symbol »#« voranstellen. Ein vergessenes »#« ist ein sehr beliebter Fehler mit schweren Folgen.

Zum Beispiel heißt

```
        move        #65, d0
```

schreibe die Konstante 65 in das Register d0. Aber

```
        move        65,d0
```

schreibt den Inhalt von Adresse 65 in das Register d0. Für Zeichenkonstanten ist die Form in Hochstrichen erlaubt. Demnach sind, wenn A den ASCII-Code 65 hat, gleichwertig

```
        move        #65,d0
und     move        #'A',d0
```

Sie sollten aber immer die letztere Form nehmen. Diese sorgt nicht nur für Klarheit, sondern auch für auf andere Computer übertragbare Programme.

Wir benutzen 3 GEMDOS-Routinen, nämlich Funktion 2 = CONOUT (Zeichen ausgeben), Funktion 1 = CONIN (Zeichen von Tastatur lesen) und Funktion 0 = TERM (Terminate and return). Für CONOUT müssen wir zuerst den Parameter (das auszugebende Zeichen) auf den Stack packen, dann folgt die Funktionsnummer (#2). Trap #1 ruft dann die Routine auf.

Der Befehl »move #'h',-(sp)« schreibt h als Wort auf den Stack. (move entspricht move.w). Das Zeichen h belegt zwar nur ein Byte, Ihr Assembler sorgt aber dafür, daß ein Wort mit einem führenden Null-Byte geschrieben wird. CONOUT gibt aber nur ein Byte aus. Der ganze Umstand hat den Grund, daß nur Worte auf den Stack »gepusht« werden können. »–(sp)« heißt, »dekrementiere sp, und zwar um 2 Adressen« (in jede Adresse paßt ein Byte,

ein halbes Wort). Der zweite Befehl »move #2,-(sp)« hat die gleiche Wirkung, nur daß jetzt die Funktionsnummer auf den Stack »gepusht« wird (»push to stack« ist korrektes Englisch).

Nun folgt der Befehl »trap #1«. Betrachten Sie das vorerst nur als »GOSUB GEMDOS« mit der Wirkung, daß dann die Routine ausgeführt wird. Das GEMDOS erwartet nun, daß wir den Stack wieder in Ordnung bringen.

Da wir insgesamt 2 Worte (also 4 Bytes) auf den Stack gepackt hatten, müssen wir nun nach dem Trap den Stackpointer um 4 erhöhen. Dazu benutzen wir den Befehl ADDQ.L (Add Quick). Im Unterschied zum normalen ADD ist dieser Befehl schneller, aber auf die Zahlen 1..8 begrenzt. Beachten Sie, daß beim Stackpointer (und Adressen allgemein) immer »lang« addiert werden muß. Die nächsten 4 Zeilen wiederholen das Spielchen mit dem Buchstaben a. Es ist Ihnen nun freigestellt, nochmals 12 Zeilen für »llo« zu tippen.

Nun folgt CONIN. Diese Funktion wartet auf eine Taste. Da hierzu nur die Funktionsnummer (#1) übergeben werden muß, haben wir nur ein Wort (2 Bytes) auf dem Stack, also »addq.l #2,sp«. Nach dem Trap steht der Code der Taste im Register D0 – interessiert hier aber nicht. Wichtig ist, daß Sie in jedem Programm auf eine Taste (oder Mausklick) warten lassen. Andernfalls kehrt es nämlich so blitzschnell zum Schreibtisch zurück, daß Sie seine Wirkung kaum beobachten können. Zum Schluß eines jeden Programms muß TERM kommen. Ich hätte anstatt

```
move  #0,-(sp)   auch    clr -(sp)
```

schreiben können.

Das heißt »clear« (lösche, lade mit Nullen) und hätte hier dieselbe Wirkung, ist aber kürzer, weshalb man es oft sieht.

4.5 Assemblieren und Linken

Wenn Sie nun dieses Programm mit einem Editor eingetippt haben, geht die Arbeit erst los. Speichern Sie den Text zum Beispiel unter dem Namen TEST.S und kehren zum Desktop zurück. Nun folgen mehrere Schritte, die von Ihrem Assembler-Paket abhängen. Lesen Sie bitte in Ihrem Handbuch nach. Zuerst müssen Sie assemblieren. Im Falle von Metacomco (bis Version 10.200) ruft man (per Mausklick) ASSEM.TTP auf und tippt dann ein

```
test.s   code test.o
```

Das heißt, assembliere »test.s« und schreibe das Ergebnis in das Code-File »test.o«. Lassen Sie das »code test.o« weg, wird nur assembliert, aber kein Objekt-File erzeugt. Das geht sehr schnell und empfiehlt sich, wenn man ein Programm nur auf Fehlerfreiheit testen will.

Haben Sie keinen Fehler gemacht (der Assembler hat nicht gemeckert), dann müssen Sie »linken«. Mein privater, heißer Tip: besorgen Sie sich »Fastlink«, den Linker, der zu ST-Pascal gehört. Der Linker ist kompakt und extrem schnell. In diesem Falle heißt die Anweisung

```
test.tos =test.o
```

4.6 Bequemer mit BATCH.TTP

Noch einfacher geht alles mit einem sog. Batch-File. Besorgen Sie sich BATCH.TTP (Public Domain, auch im ST-Entwicklungspaket). Nun erstellen Sie mit einem Texteditor ein File, in dem steht

```
assem %1.s code %1.o   opt q
fastlink %1.tos=%1.o
wait
```

Dieses File speichern Sie unter dem Namen ASS.BAT.

In dem o.g. Text steht »%1« als Variable, hier setzt BATCH.TTP später den Text ein, den Sie übergeben. »wait« ist ein kleines Programm (wait.prg), das Sie sich auch besorgen sollten. Es wartet nur auf einen Tastendruck, das können Sie auch selber schreiben, wenn Sie im Bild 4.1 »Hallo« durch »ich warte auf Taste« ersetzten (geht aber noch einfacher, kommt bald). Ist alles so weit (test.s, assem.prg, link.prg, batch.ttp, ass.bat und wait.prg sind auf der Diskette), dann klicken Sie BATCH.TTP an und geben ein

```
ass test
```

Beachten Sie: nicht »test.s« tippen, denn ».s« setzt der Batchprozessor schon ein.

WAIT.PRG hat nur die Aufgabe, auf einen Tastendruck zu warten, so daß Sie sich den Schirminhalt noch ansehen können. Das sollten Sie tun, um eventuelle Fehlermeldungen zu sehen. Nach erfolgreicher Assemblierung können Sie TEST.TOS anklicken, es sollte laufen.

Kehrt Ihr Programm hingegen dann blitzschnell zum Schreibtisch zurück und/oder es erscheinen nur Bomben, dann haben Sie etwas falsch gemacht, was der Assembler nicht erkennen konnte. Beliebte Fehler sind falsche Typen bei der Parameterübergabe oder ein Fehler bei der Stackkorrektur. In beiden Füllen steht dann eine falsche Return-Adresse auf dem Stack, was immer üble Folgen hat.

4.7 »Print Hallo«, Version 2

Nun ist es sicherlich nicht die richtige Methode, lange Strings so Zeichen für Zeichen, sozusagen als einzelne Buchstaben auszugeben, wie wir das bisher getan haben.

Mit Bild 4.2 gehen wir deshalb einen kleinen Schritt weiter.

```
* A2 mein zweites Programm

        lea       str,a1        ; Adresse von str -> a1
        move      (a1)+,-(sp)   ; das h auf den Stack
        move      #2,-(sp)      ; Funktion CONOUT
        trap      #1            ; GEMDOS aufrufen
        addq.l    #4,sp         ; Stack korrigieren

        move      (a1)+,-(sp)   ; das a auf den Stack
        move      #2,-(sp)      ; Funktion CONOUT
        trap      #1            ; GEMDOS aufrufen
        addq.l    #4,sp         ; Stack korrigieren

; u.s.w. mit l l o

; jetzt warten wir auf einen Tastendruck

        move      #1,-(sp)      ; Funktion CONIN
        trap      #1            ; GEMDOS aufrufen
        addq.l    #2,sp         ; Stack korrigieren

; nun beenden wir das Programm

        move      #0,-(sp)      ; Funktion TERM(inate)
        trap      #1

        data
str     dc.b      ' h a l l o '

        end                     ; Anweisung an den Assembler
```

Bild 4.2: Der erste Schritt in Richtung Strings

4.8 Ausgabe von Strings

Gehen wir in die vorletzte Programmzeile von Bild 4.2, da steht

```
str        dc.b      hallo
```

Wichtig ist die Assembler-Direktive »dc.b«. Bitte beachten Sie: das ist eine Anweisung an den Assembler, kein 68000-Befehl. »dc« heißt »define constant« (definiere Konstante), »dc.b« heißt dann Konstante vom Typ Byte. »str« ist ein Label, und die ganze Anweisung an den Assembler lautet nun: Setze ab (symbolischer) Adresse str die Zeichenfolge »hallo« ein. Ja, und »hallo« wollen wir nun ausdrucken. Dazu benötigen wir einen Zeiger, der auf » h a l l o « (genau: zuerst auf h) zeigt. Dazu ernennen wir das Register a1. Damit a1 mit der Adresse von »hallo« geladen wird, startet das Programm mit

```
lea     str,a1
```

Das heißt »Lade Effektive Adresse« von str in das Register a1«. Dieses zeigt nun auf str. Übrigens, Adressen sind immer lang, die falsche Schreibweise »lea.l« verzeihen aber die meisten Assembler. Häufig sieht man aber auch die Schreibweise »move.l #str,a1«. Diese ist funktional gleichwertig, dann müssen Sie aber ».l« schreiben. Wir wollen das erste Zeichen wieder mit CONOUT ausgeben und schreiben nun anstatt

```
        move    #'h',-(sp)      im vorigen Listing
        move    (a1)+,-(sp)
```

»(a1)« heißt »Inhalt der Adresse, auf die a1 zeigt«, das »+« bedeutet: »inkrementiere a1 nach diesem Befehl«. (siehe ARI im Kapitel 3). Der Rest ist Ihnen bekannt. Da a1 nun schon auf das nächste Wort (das a) zeigt, können wir das Spielchen fortsetzen, wenn Sie wollen, auch noch mit » l l o «.

Ist Ihnen aufgefallen, daß zwischen den Buchstaben von » h a l l o « Leerstellen stehen? Das war Absicht, denn die Funktion CONOUT erwartet ja immer ein Wort. Hätte ich »move.b (a1)+,-(sp) geschrieben, hätte das der Assembler zwar akzeptiert, nicht aber das GEMDOS (probieren Sie es, das gibt einen schönen Absturz).

4.9 Typwandlung muß sein

Diese lange Schreibweise ist natürlich unpraktisch. Eine mögliche Abhilfe, und gleich ein typisches Beispiel für so ein »type casting« (Typ-Anpassung erzwingen), zeigt Bild 4.3

```
* A3 mein drittes Programm

        lea     str,a1          ; Adresse von str -> a1
        move.b  (a1)+,d0
        move    d0,-(sp)        ; das h auf den Stack
        move    #2,-(sp)        ; Funktion CONOUT
        trap    #1              ; GEMDOS aufrufen
        addq.l  #4,sp           ; Stack korrigieren

        move.b  (a1)+,d0
        move    d0,-(sp)        ; das a auf den Stack
        move    #2,-(sp)        ; Funktion CONOUT
        trap    #1              ; GEMDOS aufrufen
        addq.l  #4,sp           ; Stack korrigieren

; usw. mit l l o

; jetzt warten wir auf einen Tastendruck

        move    #1,-(sp)        ; Funktion CONIN
        trap    #1              ; GEMDOS aufrufen
        addq.l  #2,sp           ; Stack korrigieren

; nun beenden wir das Programm
```

```
        move    #0,-(sp)        ; Funktion TERM(inate)
        trap    #1

        data
str     dc.b    'hallo'

        end                     ; Anweisung an den Assembler
```

Bild 4.3 »Type Casting« von Byte in Word

Gegenüber Bild 4.2 gibt es nur 3 Änderungen, aber die sind entscheidend. Der Text heißt jetzt »hallo« (ohne Blanks) und neu ist die zweite Zeile, nämlich »move.b (a1)+,d0«. Nun steht das Zeichen als letztes Byte in d0. Mit der folgenden Anweisung »move d0,-(sp)« geht das niederwertige Wort von d0 auf den Stack. Führen Sie sich immer das Registermodell vor Augen.

Stehen in einem Register die vier Bytes

```
B3 B2 B1 B0
```

dann »moved«

```
move.b                  B0
move oder move.w        B1, B0
move.l                  B3, B2, B1 B0
```

4.10 Besser mit Schleifen, zuerst »IF THEN«

In Bild 4.3 tun wir immer wieder (für jedes Zeichen) das gleiche. Grund genug, uns langsam einer effektiveren Technik für Wiederholungen, nämlich den Schleifen, zuzuwenden. Soll eine Schleife nicht endlos laufen, brauchen wir eine Abbruchbedingung. Typisch für Strings (in C und im TOS) ist, daß sie mit einem Null-Byte enden. Folglich arbeitet ein Programm, das einen »zero terminated« String ausgeben soll so, wie es Bild 4.4 zeigt:

```
* A4 mein viertes Programm

        lea str,a1              ; Adresse von str -> a1
loop
        move.b  (a1)+,d0
        cmpi.b  #0,d0           ; ist es das 0-Byte?
        beq     fertig          ; wenn ja, -> fertig
        move    d0,-(sp)        ; Zeichen auf den Stack
        move    #2,-(sp)        ; Funktion CONOUT
        trap    #1              ; GEMDOS aufrufen
        addq.l  #4,sp           ; Stack korrigieren
        bra     loop            ; branch nach loop
```

```
; jetzt warten wir auf einen Tastendruck
fertig
        move      #1,-(sp)      ; Funktion CONIN
        trap      #1            ; GEMDOS aufrufen
        addq.l    #2,sp         ; Stack korrigieren
; nun beenden wir das Programm
        move #0,-(sp)           ; Funktion TERM(inate)
        trap #1
        data
str     dc.b      'hallo'
        dc.b      0             ; ein Nullbyte markiert
                                ; String-Ende !!!!!!!!!
end                             ; Anweisung an den Assembler
```

Bild 4.4: Stringausgabe in einer Schleife

Beginnen wir wieder beim Ende, so sehen Sie, daß dem »dc.b hallo« nun noch ein »dc.b 0« folgt. Es ist übrigens durchaus erlaubt (und üblich) die beiden Anweisungen in einer Zeile als »dc.b hallo,0« zu schreiben. Zum Anfang des Listings von Bild 4.4: Neu ist die Marke »Loop«. Danach holen wir uns wieder ein Byte mit »move.b (a1)+,d0« und nun kommt etwas Neues. Der Befehl

```
cmp heißt compare (vergleiche)
```

»cmpi« ist eine Abwandlung davon, wobei »i« für »immediate« (direkt) steht, und mit direkt ist eine Konstante gemeint. Die meisten Assembler setzen sozusagen das »i« automatisch ein, wenn Sie es vergessen, aber wir sind ja korrekt. Den cmp-Befehl müssen Sie bei den Operanden arabisch lesen (von rechts).

Also heißt »cmpi #0,d0«: vergleiche d0 mit 0.

Der Befehl selbst setzt nur Flags (Bits) im CCR (Condition Code Register), was wir später noch ausführlich behandeln. Vorerst ist wichtig, daß es Befehle gibt, die auf diese Flags reagieren. Einer dieser Befehle heißt

```
beq      branch if equal
springe  wenn gleich
```

Hier wäre es also egal, in welcher Reihenfolge Sie die Operanden lesen, aber es gibt auch Befehle wie zum Beispiel »bhi« (branch if higher, springe wenn größer als), und da würde dann »bgt #0,d0« heißen »springe wenn d0 > 0«. Nun, was machen wir, wenn wir ein Null-Byte gefunden haben? Wir sind fertig mit der Schleife, also springen wir zum Label »fertig«. Wenn nicht, wird der nächste Befehl ausgeführt, und den und die folgenden kennen Sie schon, bis auf einen:

```
bra heißt branch always (springe immer)
```

Dieses »bra« ist die einfachste bedingte Verzweigung, es entspricht dem GOTO in anderen Sprachen. Mit »bra loop« greifen wir also die Schleife wieder auf.

4.11 String-Ausgabe mit GEMDOS #9

Damit hätten wir schon eine ganz nette Technik, Strings auszugeben, aber eigentlich wollte ich Ihnen nur ein paar Grundtechniken beibringen. Für die Stringausgabe selbst gibt es nämlich eine GEMDOS-Funktion, die mit der Nummer 9. Bild 4.5 zeigt, wie man Nummer 9 praktisch anwendet.

```
* A5 mein fuenftes Programm
        pea     str         ; Adresse von str auf den
                              Stack
        move    #9,-(sp)    ; Funktion 'Print Line'
        trap    #1          ; GEMDOS aufrufen
        addq.l  #6,sp       ; Stack korrigieren
; jetzt warten wir auf einen Tastendruck
fertig
        move    #1,-(sp)    ; Funktion CONIN
        trap    #1          ; GEMDOS aufrufen
        addq.l  #2,sp       ; Stack korrigieren
; nun beenden wir das Programm
        move    #0,-(sp)    ; Funktion TERM(inate)
        trap    #1
        data
str     dc.b    'hallo'
        dc.b    0           ; ein Nullbyte markiert
                            ; String-Ende !!!!!!!!!
        end                 ; Anweisung an den Assembler
```

Bild 4.5: Ausgabe von Strings mittels der GEMDOS-Funktion 9

Alle TOS-Funktionen erwarten ihre Parameter auf dem Stack. Im Falle eines Strings von zum Beispiel 1000 Zeichen (ganz normal, eine Bildschirmmaske) wäre das nun sehr unpraktisch. Deshalb werden solche Parameter »by reference« übergeben, soll heißen, man übergibt ihre Adresse. Eine Adresse auf den Stack zu packen kommt sehr häufig vor, weshalb es dafür einen speziellen Befehl gibt, nämlich

```
pea = Push Effective Adress
```

Effektive Adresse deshalb, weil die Adresse auch aus einem Ausdruck erst errechnet sein kann, nicht muß (siehe Kaptitel 3). Wie auch immer, mit »pea str« geht die Adresse von »str« auf den Stack, mit dem nächsten Befehl folgt ihr die Funktionsnummer und dann »trappe« wie gehabt. Beachten Sie bitte hier: Wir haben eine Adresse (4 Byte) und ein Wort auf den Stack (2 Byte) gebracht, müssen also schon 6 addieren, um den Stackpointer zu restaurieren. Übrigens: häufig findet man Listings, in denen anstatt

```
pea str
move.l    #str,-(sp)
```

steht. Dahinter steckt wohl das Motto, warum ~~so~~ einfach, wenn es auch umständlicher geht. Aber eines können wir daraus lernen: »#« heißt nicht nur Konstante, sondern auch »Wert von/Adresse von«. Würde man beim

```
move.l    #str,-(sp)
```

das »#« vergessen, passierte Übles. Es geht nicht die Adresse von str auf den Stack, sondern der Inhalt der 4 Bytes ab Adresse str. Die Funktion Nummer 9 faßt dieses Langwort natürlich als Adresse auf und druckt nun alles, was da kommt, bis (hoffentlich) irgendwann einmal ein Null-Byte gefunden wird.

4.12 Die erste DBcc-Schleife

Ich möchte Ihnen nun noch eine letzte Möglichkeit vorstellen, Texte auszugeben, und dabei natürlich wieder etwas ganz anderes lehren, nämlich wie man eine Zählschleife (ähnlich dem FOR NEXT in Basic) realisiert. Wir wollen die Buchstaben A bis Z drucken und zwar in der Art, wie man es als Basic-Programm schreiben würde

```
10 FOR I= ASC("A") TO ASC("Z")
20 PRINT CHR$(I)
30 NEXT
```

Bild 4.6 bringt die Lösung.

```
* A6 die DBcc-Schleife
* Buchstaben A..Z drucken
        move      #25,d1        ; der Schleifenzaehler
        move      #'A',d2       ; Anfangszustand
loop    move      d2,-(sp)      ; Buchstabe auf Stack
        move      #2,-(sp)      ; Funktion CONOUT
        trap      #1            ; GEMDOS aufrufen
        addq.l    #4,sp         ; Stack korrigieren
        addq      #1,d2         ; naechster Buchstabe
        dbra      d1,loop       ; Siehe Text!
```

```
;jetzt warten wir auf einen Tastendruck
        move    #1,-(sp)        ; Funktion CONIN
        trap    #1              ; GEMDOS aufrufen
        addq.l  #2,sp           ; Stack korrigieren
;nun beenden wir das Programm
        move    #0,-(sp)        ; Funktion TERM(inate)
        trap    #1
        end                     ; Anweisung an den Assembler
```

Bild 4.6: Drucken von A bis Z mit DBcc

Im Gegensatz zu den meisten Konkurrenten hat der 68000 einen Schleifenbefehl schon eingebaut, nämlich:

```
DBcc Dn,Marke
```

Das steht für »Decrement and Branch on Condition Code«. Wow, das ist ein Ding! Also der Reihe nach: Mit dem DBcc-Befehl wird immer ein Datenregister angegeben, das kann D0 bis D7 sein, nennen wir es Dn. Vor dem Eintritt in die Schleife wird Dn ein Wert zugewiesen. In der Schleife, genau, immer dann, wenn der DBcc-Befehl durchlaufen wird, wird Dn um Eins dekrementiert. Solange Dn dabei nicht -1 wird, erfolgt ein Sprung zu Marke, ansonsten wird der nächstfolgende Befehl ausgeführt. Nun zum »cc«: Zusätzlich kann man nun noch vor dem DBcc-Befehl mit zum Beispiel einer CMP-Anweisung eine Bedingung testen und dann zum Beispiel sagen:

```
CMP     (A0)+,D0
DBeq    D1,Marke
```

In diesem Falle erfolgt der Sprung zu »Marke« nur dann, wenn die Bedingung »A(0) eq (equal = gleich) D0« nicht erfüllt ist, ansonsten wird die Schleife beendet. Man kann es auch so sehen: Die Schleife wird durchlaufen, solange die cc-Bedingung erfüllt ist, aber höchstens, solange wie der Zähler noch nicht auf -1 ist. Die Kürzel für »cc« sind die gleichen wie beim bcc-Befehl. Z. B. gibt es BEQ (Branch if Equal) und DBEQ (Decrement and Branch if Equal). Die Einzelheiten zu allen »CCs« finden Sie im nächsten Kapitel, machen wir aber erst mal mit der Praxis weiter. Häufig interessiert nämlich die Bedingung überhaupt nicht, man will nur zählen. In diesem Fall sagt man einfach

```
DBRA,
```

was »Decrement and Branch Always« (springe immer) heißt, natürlich nur solange der Zähler nicht abgelaufen ist. Häufig sieht man auch »DBF«, wobei F für »False« (Falsch) steht, das ist nur eine andere Schreibweise. Gute Assembler akzeptieren sowohl »RA« als auch »F«. Nun können wir uns dem Listing von Bild 4.6 zuwenden. Wir wollten ja die 26 Buchstaben von A bis Z drucken. Weil der Zähler D1 aber immer bis -1 läuft, initialisiere ich ihn mit 25, siehe erste Zeile. Den Code für Buchstaben halte ich im Register D2, das wird also zuerst mit »A« geladen.

Bei »Loop« geht es nun los. Wie gehabt, drucken wir ein Zeichen mittels der Funktion CONOUT, also erst das Zeichen auf den Stack (D2), dann die Funktionsnummer hinterher und »trappe«. Auch der Stack wird wieder korrigiert, und nun kommt das Neue. Mit »addq #1,d2« wird d2 inkrementiert, aus dem A wird also ein B (dann aus dem B ein C u.s.w.). Die Arbeit leistet die nächste Zeile.

```
        dbra    d1,loop
```

heißt: Dekrementiere d1, wenn es dann noch nicht -1 ist, springe zu »Loop«, ansonsten nächster Befehl. Hier ginge es also im Falle von -1 bei »warte auf Tastendruck« weiter.

4.13 Eingabe von Strings

Da so ein typischer User seine Wünsche dem Programm in der Form von Texten mitteilt (auch wenn ein Basic-Programmierer FOR tippt, ist das nur ein Text aus Sicht von uns Assembler-Programmierern), müssen wir uns wohl mit diesem Thema befassen. Wenn ich Ihnen nun sage, »der eingetippte Text geht in einen Puffer«, müßten Sie mir das glauben, also definiere ich die Aufgabe visuell nachvollziebar, und zwar so: Das Programm soll fragen »Wie heißt Du?«. Der User gibt dann einen Text ein (ich hoffe, seinen Namen), und das Programm antwortet dann »Guten Tag, Name«. Kleine Zusatzforderung: Mehr als 78 Zeichen dürfen nicht akzeptiert werden; sie sollen abgeschnitten werden, weil unser Puffer nicht größer ist. Die Lösung finden Sie in Bild 4.7.

```
*A7 Zeile lesen, Text ausgeben

loop                            ; Endlos-Schleife
                                ; Abruch mit Control_C
        lea frage,a0            ; Adresse 'Wie heisst Du? '
        bsr print               ; Unterprogramm aufrufen

; nun eine Zeile lesen:

        move.b #80,puffer       ; max. Lange Eingabe
        pea puffer              ; Pufferadresse -> Stack
        move #10,-(sp)          ; Funktion READ LINE
        trap #1                 ; GEMDOS aufrufen
        addq.l #6,sp            ; Stack koorigieren
        move d0,d1              ; merke Ist-Laenge

        lea antwrt,a0           ; Adresse 'Guten Tag'
        bsr print               ; und drucken
        lea puffer,a0           ; Adresse Puffer
        move.b #0,2(a0,d1)      ; markiere String-Ende
        addq.l #2,a0            ; auf Begin Text
        bsr print               ; und drucken

        bra loop                ; auf ein Neues
```

```
print   pea (a0)                ; Adresse Text-String
        move #9,-(sp)           ; Funktion PRINT LINE
        trap #1                 ;
        addq.l #6,sp
        rts
        data
frage   dc.b    13,10,'Wie heisst Du? ',0
        ds.w    0
antwrt  dc.b    13,10,'Guten Tag, ',0
        bss
puffer  ds.b    80
        end
```

Bild 4.7: Eingabe und Ausgabe von Strings

Zuerst: Das Programm läuft in einer Endlosschleife. Diese Schleife können Sie über Control-C (Tasten Control und C gleichzeitig drücken) verlassen. Das ist ein Feature des TOS, es zu verhindern, kostet zusätzlichen Programmieraufwand. Beginnen wir im Listing wieder am Ende. Da steht

```
puffer   ds.b  80
```

»ds« heißt »define storage« (definiere Speicher). Das ist eine Anweisung an den Assembler (Direktive). Das »b« bedeutet wieder Byte. Die ganze Anweisung bewirkt, daß der Assembler einen Speicherbereich von 80 Bytes reserviert und diesem die Adresse »puffer« gibt. Nun brauchen wir 2 Texte, nämlich einen der Art »Wie heißt Du«, einen zweiten als »Guten Tag«. Der Trick ist nun, daß ich jeden Text mit den Zeichen 13 und 10 beginnen lasse. 13 heißt CR (Carriage Return) und 10 LF (Line Feed). Die GEMDOS-Funktion Nummer 9 interpretiert diese Steuerzeichen richtig, sprich, sie fängt eine neue Zeile an.

4.14 Programm-Segmente Text, Data und BSS

Wichtig sind nun noch diese Direktiven:

```
text
data
bss
```

»data« ist die Anweisung an den Assembler, die folgenden Daten in das Datensegment des Programms zu packen. Dazu müssen Sie wissen: Ein TOS-Programm besteht aus Segmenten. Das erste Segment heißt Text oder Code. Darin steht das eigentliche Programm. Sie können, in manchen Assemblern müssen Sie sogar, Ihr Programm mit dem Wort »text« starten.

Im Data-Segment stehen alle initialisierten Daten, also solche, die einen Wert haben, wie zum Beispiel unsere Texte. Im »bss« (block storage segment) werden Daten abgelegt, die erst während der Programmlaufzeit entstehen. Praktisch sind es nur reservierte Speicherbereiche.

Nun beachten Sie bitte noch das »ds.w 0« zwischen »frage« und »antwrt«. Sie wissen noch, beim 68000 ist nur der Zugriff auf gerade Adressen erlaubt (Ausnahme »kriegen wir später«). Da ich nicht weiß bzw. zu faul bin, abzuzählen, wieviele Bytes der Text »Wie heißt Du?« hat, der nächste String aber auf einer geraden Adresse beginnen muß, schreibe ich »ds.w 0«.

Das heißt eigentlich ziemlich sinnlos »reserviere Speicher von null Worten«. Der Assembler justiert aber bei einem »ds.w« auf die nächste Wortgrenze, sprich, er stellt seinen internen Zeiger auf eine gerade Adresse, sofern der nicht ohnehin schon da steht. Ein »ds.w« zuviel schadet also nichts. »ds.w« nehme ich, weil das jeder Assembler kann. Oft genug gibt es dafür aber auch die spezielle Direktive EVEN.

Merke: DS.W 0 und EVEN sind gleichwertig.

4.15 Unterprogramme

Wir haben zwar mit der GEMDOS-Funktion Nr. 9 schon einen recht bequemen Weg, Texte auszugeben, aber auch das sind dann immer 4 Befehle. Da in einem Programm sehr viele Texte auszugeben sind, ist es an der Zeit, auch das zu rationalisieren, und das geschieht am besten mit Unterprogrammen.

In Basic sagt man dazu GOSUB Zeile (oder GOSUB Marke), in Assembler heißt das BSR Marke (Branch to Subroutine) oder JSR (Jump to Subroutine). Der Unterschied zwischen BSR und JSR ist, daß BSR auf die Weite von »nur« 64 K begrenzt ist, JSR hingegen über den vollen Adreßbereich geht. In unseren kleinen Beispielen von nur so 1-10 K reicht BSR allemal.

Wie in Basic muß auch hier ein Unterprogramm mit Return enden, was man mit RTS (Return from Subroutine) abkürzt. Bliebe noch das Schwierigste: einem Unterprogramm muß man in aller Regel Parameter übergeben. Wir haben hier ein Unterprogramm namens »print«, und diesem müssen wir sagen, was es »printen« soll. Da es sich um Strings handelt, übergebe ich die Adresse des jeweiligen Strings.

Vereinbart ist (zwischen mir und dem ST), daß die Adresse im Register a0 übergeben wird. Da das Programm mit dem Text unter Marke »frage« starten soll, schreibe ich als erste Zeile »lea frage,a0« (lade Adresse von Frage in das Register a0) und sage dann einfach »bsr print«.

»print« arbeitet mit der schon bekannten GEMDOS-Funktion 10, dieser muß ich also A0 auf den Stack packen. Nun passen Sie auf, es kommt ein beliebter Fehler. Mit »lea frage,a0« habe ich die Adresse von »frage« in das Register A0 geladen. A0 zeigt also auf »frage«. Deshalb darf man im Unterprogramm »print« nicht »pea a0« schreiben, sondern richtig »pea (a0)«. In Klammern heißt immer indirekt, also »Inhalt von... / Zeiger auf... Nun, den Rest von »print« kennen Sie schon aus Bild 4.5; wenden wir uns dem Neuen zu.

4.16 String-Eingabe mit GEMDOS #10

Zur Eingabe eines Strings benutzt man am besten die GEMDOS-Funktion Nummer 10. Diese erwartet nur einen Parameter, nämlich die Adresse des Puffers, in den der String eingelesen werden soll. Leider hat der Puffer ein besonderes Format, nämlich

Byte 0: Maximal erlaubt Anzahl Zeichen
Byte 1: Tatsächlich eingebene Anzahl Zeichen
Byte 2 bis (Byte 1): Der String an sich

Beachten Sie bitte, daß ich ab Null zähle, dieses sollte man aus einem Grund bei Puffern immer tun. Es gibt die Adressierungsart ARI mit Offset. Wenn zum Beispiel A0 der Zeiger auf den Pufferbeginn ist, dann kann ich mit »0(a0)« auf Byte 0 zugreifen, mit »1(a0)« auf Byte 1 usw. O.K., nun laßt uns eine Zeile lesen. Mit der dritten Anweisung in Bild 4.7 geht es los. »move.b #80,puffer« schreibt die Maximallänge in das erste Byte und das falsch!! Ich habe hier absichtlich einen Fehler eingebaut, der vielleicht jahrelang nie auffällt oder nur manchmal auftritt, und keiner weiß warum. Mit »#80« erlaube ich die Eingabe von 80 Zeichen. Die Eingabe startet aber im Puffer beim dritten Byte (relativ Byte 2). Gibt also wirklich jemand 80 Zeichen ein, so schreibt er über den Puffer hinaus. Hier passiert nichts, aber stellen Sie sich vor, Sie hätten in Ihrem Programm noch eine Zeile nach der Marke Puffer stehen, zum Beispiel als

```
Puffer_2  ds.b  20,
```

wo Sie andere Daten schon abgelegt haben. Dann würde doch ein zu langer Name diese Daten überschreiben, peinlich, peinlich ...

Ändern Sie also die #80 in 78 oder das »ds.b 80« in »ds.b 82« und – langer Rede kurzer Sinn: Basic und andere Hochsprachen reservieren immer die richtige Länge für die einzelnen Variablen, in Assembler müssen Sie sich darum selbst kümmern. Faustregel: im Zweifelsfall immer etwas mehr.

Weiter im Listing: Nach dem Aufruf von »Nummer 10« steht die Ist-Länge des Strings im Byte 1, aber auch im Register D0. Da alle GEMDOS-Funktionen einen Wert (oder Fehlercode) im Register D0 zurückgeben, dieses Ergebnis also beim nächsten GEMDOS-Aufruf weg wäre, retten wir es mit »move d0,d1«. Nun haben wir den Namen und starten in bekannter Art die Ausgabe von »Guten Tag« mittels des Unterprogramms »print«.

4.17 Text ins String-Format

Dem »Guten Tag « soll der Name folgen, der steht im Puffer, und nun wird es schwierig. Damit der Name ein echter TOS-String wird, muß er mit einem Null-Byte enden. Leider tut uns die GEMDOS-Funktion nicht den Gefallen, das Null-Byte in den Puffer zu schreiben, das müssen wir schon selber tun. Ich habe schon einige Listings gesehen, wo das fehlte. Daß die Programmierer den Fehler nicht gemerkt haben, hat einen einfachen Grund. Beim Programmstart löscht TOS den Speicher, schreibt also überall Null-Bytes hin. Folglich klappt die Sache beim ersten Mal mit Sicherheit, sogar auch noch, wenn beim zweiten Versuch der neue Text länger ist als der erste. Schief geht es, wenn der neue

Text kürzer ist als der Vorgänger, dann wird ein Mischmasch gedruckt. Also machen wir es richtig und erledigen das mit einem einzigen Befehl. So etwas geht nur beim 68000 so elegant. Wenn Sie also in Kapitel 3 Probleme hatten, sich vorzustellen, wofür man »ARI mit Index und Offset« wohl brauchen könnte, hier hätten wir schon einen Fall. Nehmen wir an, der Name hieße Anton. Dann steht im Puffer

unter Byte	0	1	2	3	4	5	6	7
das Zeichen	#80	#5	'A'	'n'	't '	'o'	'n'	?

Auf Puffer plus 7 müssen wir also das Nullbyte schreiben. Dank des Befehls »lea puffer, a0« zeigt a0 auf Byte 0. Schon vorher hatten wir die Länge in d1 gerettet. Also bewirkt

```
move.b      #0,2(a0,d1)
soviel wie  (a0) plus 2 plus d1
oder        (a0) plus 2 plus 5
oder        (Beginn Puffer plus 7)
```

In das so adressierte Byte wird also ein Null-Byte (#0) geschrieben.

Nun folgt noch ein »addq.l #2,a0«, womit a0 auf das Byte 2 (den ersten Buchstaben des Namens) zeigt, und das war's schon, fast ... Mit »bsr print« können wir den Namen ausgeben, mit »bra loop« startet das Programm wieder, es sei denn, Sie stoppen es mit Control-C.

Kapitel 5

Verzweigungen und Menü-Technik

In diesem Kapitel geht es nochmals
um das »IF THEN« in Assembler
und um das leidige, aber sehr notwendige
Bit-Schieben.
Natürlich kommt auch wieder
die Praxis an die Reihe.
Diesmal lernen wir den Einsatz
der Funktionstasten
und das Prinzip von »ON X GOSUB«.

5.1 »IF THEN« im Detail

Wir hatten es ja schon angewendet, nun schauen wir es uns etwas genauer an, das »IF Bedingung THEN GOTO«. Prinzipiell funktioniert es so wie in den Hochsprachen: Man fragt eine Bedingung ab und verzweigt in Abhängigkeit vom Ergebnis. Der kleine Unterschied zu den Hochsprachen: Die Bedingung ist der Zustand einiger Bits im CCR (Condition Code Register), das wiederum Teil des Statusregisters ist. Wie dieses Register aussieht, zeigt das folgende Bild (Bild 5.0).

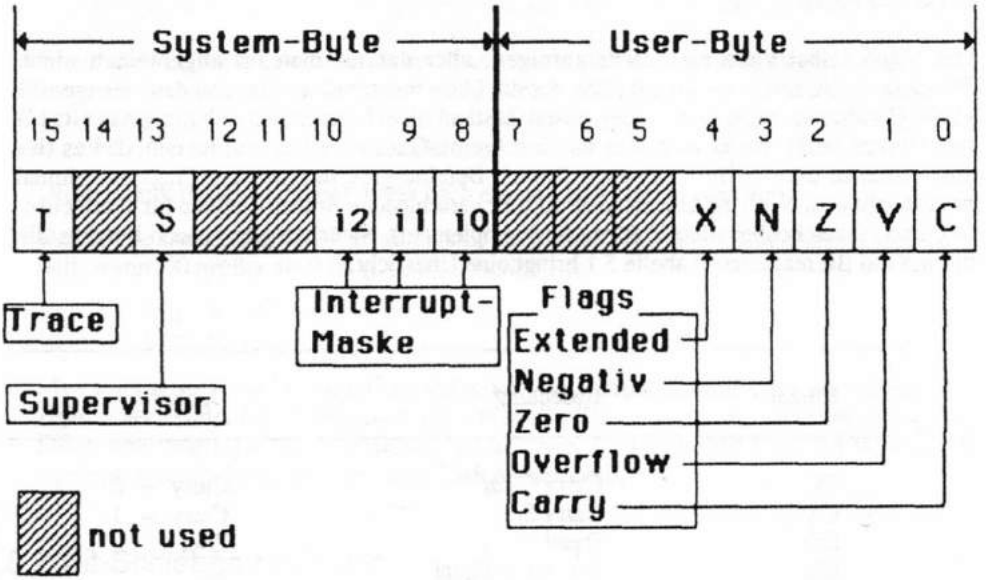


Bild 5.0: Das Status-Register des 68000

5.1.1 Das Statusregister

Sie sehen sofort, daß das Wort in ein System- und ein User-Byte unterteilt ist. Dazu merken Sie sich bitte vorerst nur, daß der 68000 zwei Betriebsarten kennt, nämlich Supervisor-Mode und User-Mode. Wir werden uns noch ausführlich damit beschäftigen, jetzt nur soviel: Der Supervisor-Mode ist dem TOS vorbehalten, wir »USER« arbeiten vorerst nur im User-Mode. Deshalb interessieren uns von den Bits (auch Flags genannt) X, N, Z, V und C.

5.1.2 Die Flags

Es gibt viele Befehle, die diese Flags beeinflussen, meistens jedoch sind es mathematische Operationen mit zwei Operanden, wobei der Quelloperand vom Zieloperanden subtrahiert wird. Wird dabei das Ergebnis negativ, setzt der 68000 das N-Flag (das Bit wird 1). Entsteht ein Überlauf, wird das Overflow-Bit gesetzt, bei einem Übertrag (Addition) bzw. beim »Borgen« während der Subtraktion geht das Carry-Flag auf Eins. Umgekehrt: Treten diese Zustände während der Operation nicht ein, werden die entsprechenden Flags auf Null gesetzt. Vorerst nur als Info: T ist das Trace-Bit, S das Supervisor-Bit und i0,i1,i2 die Interrupt-Maske.

5.1.3 Die Abfrage der Flags

Die Flags selbst kann man zwar abfragen, aber das tut man im allgemeinen nicht. Stattdessen schreibt man einen Befehl, der die Flags beeinflußt (prüft) und dann sinngemäß »GOTO Adresse, wenn dieses Flag diesen Zustand hat«. Merken Sie sich bitte, daß GOTO hier Branch heißt, wofür man aber nur B schreibt. Ganz entscheidend ist nun, daß es (im Gegensatz zu den »8-Bittern«) auch Branch-Befehle gibt, die mehrere Flags auf einmal berücksichtigen. Noch ein Unterschied: Es gibt verschiedene Branch-Befehle für vorzeichenlose und vorzeichenbehaftete Zahlen (2er-Komplement). Natürlich gibt es auch Befehle, die nur auf ein Bit reagieren. Tabelle 5.1 bringt eine Übersicht.

	Kürzel	Bedeutung	Deutsch
	CC	Carry Clear	Carry = 0
	CS	Carry Set	Carry = 1
	EQ	Equal	Z = 1
***	GE	Greater or Equal	>=
***	GT	Greater Than	>
	HI	Higher	>
***	LE	Less or Equal	<
	LS	Less or Same	<=
***	LT	Less Than	<
	MI	Minus	-
	NE	Not Equal	<>
	PL	Plus	+
***	VC	oVerflow Clear	V = 0
***	VS	oVerflow Set	V = 1

Tabelle 5.1 Kürzel der »Condition Codes«

Die in der Tabelle mit *** gekennzeichneten Operatoren gelten für Zahlen im sog. 2er-Komplement-Format, das sind die, bei denen auch negative Werte erlaubt sind. Die Befehle fangen immer mit B (wie Branch) an, gefolgt von 2 Buchstaben, die die Kurzform für die Bedingung sind. Wenn Sie nun zum Beispiel BEQ (springe, wenn gleich) schreiben, dann hängt es ausschließlich vom Z-Flag ab, ob der Befehl ausgeführt wird oder nicht. Das Z-Flag kann aber durch einen mehr oder weniger weit vor dem BEQ liegenden Befehl beeinflußt worden sein. Wenn Sie nun genau wissen, welcher Befehl das Z-Flag wie beeinflußt, dann könnten Sie das Risiko eingehen. Sicherer ist es auf jeden Fall, direkt vor BEQ einen Befehl zu schreiben, der das prüft. Wenn ich zum Beispiel springen will, wenn das Register D0=0 ist, dann schreibe ich:

```
CMP   #0,D0
BEQ   Marke
```

Der CMP-Befehl subtrahiert den Quelloperanden vom Zieloperanden, ändert je nach Ergebnis die Flags, schreibt aber nicht das Ergebnis ins Ziel. D. h., der Vergleichsbefehl wirkt auf die Flags wie eine Subtraktion. Das müssen sich die armen Kollegen mit den »8-Bittern« immer vor Augen führen, wenn Sie danach einzelne Flags (mit je einem Befehl) testen. Sie haben es besser. Sie dürfen zum Beispiel schreiben BGE (Springe, wenn größer oder gleich). Sie müssen sich nur dreierlei merken:

1. Diese Luxus-Befehle sind nur direkt nach einem CMP korrekt wirksam.
2. Der zweite Operand wird gegen den ersten verglichen. Wenn ich zum Beispiel springen will, wenn D0 größer als 9 ist (D0>9), schreibe ich:

   ```
   CMP #9,D0
   BGT Marke
   ```

3. Man muß wissen, ob man die Operanden als vorzeichenbehaftete oder vorzeichenlose Zahlen verwendet hat. Sinngemäß kann man die Kürzel auch im Zusammenhang mit DBcc anwenden, DBMI oder DBGT wären zwei Beispiele. BRA ist ein Sonderfall (springe immer); dem entspricht auch DBRA oder DBF.

5.2 Bit-Schieben muß sein

Mit Bild 5.1 komme ich wieder zur Praxis. Die Funktion CONIN gibt in D0 den Code der gedrückten Taste zurück und zwar als Langwort. Nun will ich wissen, welche Taste welchen Code erzeugt. Dazu muß ich den Inhalt von D0 ausdrucken und zwar in hex. Hauptaufgabe von Bild 5.1 ist somit die Ausgabe eines Langwortes, wie es in einem Register steht, als sog. Hex-String.

```
; Demo Hex-Ausgabe
start   move.w   #1,-(sp)     ; Funktion CONIN
        trap     #1           ; aufrufen
        addq.l   #2,sp        ; Stack korrigieren
        cmpi.b   #'x',d0      ; x eingeben
        beq      stop_it      ; wenn ja, stop
        move.l   d0,d2        ; Code nach d2 kopieren

; print (d2) in hex
; -----------------
        move     #7,d1        ; Langwort hat 8 Nibbles
next    rol.l    #4,d2        ; Hole ein Nibble
        move.l   d2,d3        ; nach d3
        andi.b   #$0f,d3      ; maskiere es
        addi.b   #48,d3       ; '0'..'9' waere jetzt OK
        cmpi.b   #58,d3         ist es >9?
        bcs      out          ; nein, dann Ausgabe
        addi.b   #7,d3        ; sonst muss es 'A'..'F' sein
out     move     d3,-(sp)     ; print via GEMDOS
        move     #2,-(sp)     ; mit Funktion CONOUT
        trap     #1
        addq.l   #4,sp
        dbra     d1,next      ; next nibble
        bra      start        ; und wieder von vorn

stop_it clr-(sp)              ; Funktion 0 = Ende
        trap     #1

        end
```

Bild 5.1: Ein Hex-Konverter

5.2.1 Ein Hex-Konverter

Das Programm zeigt recht gut, daß man in Assembler oft auf Bit-Ebene arbeiten sollte. Damit läßt sich zum Beispiel die Hex-Konvertierung wesentlich einfacher lösen als mit der klassischen Methode (fortlaufende Divison durch 16). Sie werden bei der Gelegenheit auch sehen, warum die Hex-Darstellung so vorteilhaft ist. Ein Beispiel: Das Langwort besteht aus den 4 Bytes des Inhalts AA, BB, CC, DD. Ich sehe sofort, daß im höherwertigen Wort AABB und im niederwertigen CCDD steht. In dezimal (2864434397) dürfte das schwerfallen. Das Problem der Routine »prthex« (Print in Hex) ist nun, daß sie tatsächlich ASCII- Zeichen ausgeben muß. Hat dort ein Digit den Wert Null, muß ich den ASCII-Code 48 (dezimal) ausgeben, um die »0« auf dem Schirm zu sehen. Das klappt ganz gut bis zur Neun (57), doch für 10 muß ich hex »A« drucken, und das hat den ASCII-Code 65,

B hat 66 usw. Diese Lücke zwischen »9« und »A« müssen wir also berücksichtigen. Zweites Problem: Die Hexzahl sei $12345678 ($ heißt hex). $1 ist ein Nibble (Halbbyte), das im Register 4 Bit belegt (0001). Natürlich muß ich $1 zuerst ausgeben, doch dafür muß ich $1 auf den Platz von $8 bringen, weil CONOUT das Zeichen (nach der Umwandlung in die ASCII-1) im niederwertigen Wort (lower Byte) sehen will. Das Byte hat aber 8 Bit, 4 übertrage ich, die übrigen 4 Bit haben Werte, die nur stören, folglich muß ich sie ausblenden. Damit ergibt sich folgender Ablauf:

1. Das Nibble $1 auf den Platz von $8 bringen
2. Dort die übrigen 4 Bit des Bytes auf Null setzen.
3. Das Nibble in ASCII wandeln.
4. Das Zeichen ausgeben.
5. Wiederhole 1. bis 4. mit den Nibbles $2, $3...$8.

Schritt 1 wird mit dem ROL-Befehl erledigt (Rotate Left). Wir benutzen von ROL die Syntax

```
ROL  #4,d2
```

Das heißt, rotiere den Inhalt von D2 um 4 Bits nach links. Und was heißt nun rotieren? Nehmen wir an, in den d2 stehen diese 32 Bit

vorher	1111 0000 0000 0000 0000 0000 0000 0000
nach ROL #4	0000 0000 0000 0000 0000 0000 0000 1111

D. h., die Bits werden nach links geschoben, und die Bits, die dann ganz links »herausfallen«, werden rechts wieder eingespeist. Anders sähe es beim ASL-Befehl (Shift left, schiebe nach links) aus. Da wird auch nach links geschoben. Rechts werden Nullen eingespeist und links fallen die Bits heraus. Nun, wir haben ROL gewählt, und damit unser Ziel erreicht. Das Nibble (so nennt man ein Halb-Byte), das vorher ganz links stand, steht nun ganz rechts.

Nun wieder zum Listing. Mittels CONIN wird ein Zeichen von der Tastatur gelesen. Damit unser Programm auch ordentlich stoppen kann, vereinbaren wir, daß nach der Eingabe von x aufgehört werden soll. Die Marke heißt übrigens nicht »stop«, sondern »stop_it«, weil »stop« ein 68000-Befehl ist. Ansonsten steht unser Zeichen im Register D0. Weil D0 mit jedem GEMDOS-Aufruf verändert wird, retten wir es in das Register D2, und nun geht es los. Die Schleife wird mit dem Befehl DBRA erledigt. Unsere Schleife soll achtmal durchlaufen werden, folglich muß ich vorab (wegen des -1) den Schleifenzähler D1 mit 7 laden. Nun erfolgt das berühmte ROL. Danach steht das Nibble am richtigen Platz, aber ausgeben kann ich leider nicht diese 4 Bits alleine, ich brauche ein Byte für ein ASCII-Zeichen, also 8 Bit.

5.2.2 Die Sache mit den Masken

Die 4 Bit links von meinem Nibble haben aber einen Wert, und der muß weg. Genau: Die 4 höherwertigen Bits des Bytes müssen 0000 werden. Das geschieht über die Maske $F mit »andi.b #$0F,d3«. Beispiel:

in D3 steht	11011	1010
AND Maske	0000	1111
ergibt	0000	1010

Das Verfahren lebt davon, daß logisch-UND nur dann wahr (1) ergibt, wenn alle Eingangsgrößen wahr sind. In Assembler wirkt UND (and) Bit für Bit. Nachdem wir so den reinen Zahlenwert (0..15 dezimal) isoliert haben, beginnt die Konvertierung in ASCII. 0 bis 9 (als Zahl) kann direkt in 0 bis 9 (als ASCII- Zeichen) umgesetzt werden, das sind die ASCII-Codes 48 bis 57. D. h. aber auch, 48 müssen wir mindestens addieren. Nun vergleichen wir D3 gegen 58 (also eine hex 10, die man als A ausgeben muß). Ist die Zahl <10 (also 0 bis 9) kann alles so bleiben, es geht zur Ausgabe. Andernfalls müssen wir noch die Lücke in der ASCII-Tabelle (65 für A-58=7) addieren.

An dieser Stelle ein Hinweis. Man sieht manchmal den Weg, daß erst geprüft wird, ob der Wert zwischen 0 und 9 liegt und dann im zweiten Test, ob es ein Wert zwischen 10 und 15 ist. Diese Methode ist OK, wenn die Zeichen aus einer Eingabe stammen, wo ja der User auch ungültige Zeichen (nicht 0..9, A...F) eintippen kann. Wir holen hier aber Zahlen aus einem Register, da kann so etwas nicht drinstehen!

Die Befehle CMPI und BCS hatten wir ja schon. Warum aber BCS? Ja, eigentlich nur, um Ihnen zu zeigen, was andere Leute so schreiben (und denken müssen), die von CPUs kommen, die nicht so komfortabel wie der 68000 sind. Ein Vergleich wirkt auf die Flags, wie eine Subtraktion. Bei einem »Borgen« wird auch das Carry-Flag gesetzt. Das ist aber der Fall, wenn D3 > 58 ist.

Wenn Sie das Programm nun laufen lassen, wird Sie sicherlich stören, daß nach einer Ausgabe kein Zeilenvorschub erfolgt. Damit hätte ich eine kleine Aufgabe für Sie: Vor dem »bra start« müßten Sie noch ein paar Befehle einfügen, die die Zeichen CR (13) und LF (10) ausgeben. Sie haben die Wahl, zweimal CONOUT aufzurufen, oder Sie deklarieren einen String mit zum Beispiel als »crlf dc.b 13,10,0« und geben dann »crlf« über die GEMDOS-Funktion 9 aus. PRINT LINE

5.3 ASCII-Code und Scan-Codes

Sozusagen als Vorübung für etwas schwierigere Dinge, die da kommen werden, wollen wir uns nun mit den Scan-Codes befassen. Dahinter steckt folgende Kleinigkeit: Der ST hat in seiner Tastatur einen extra Mikro-Computer, der u. a. die Tastatur überwacht. Dieser Mikro fragt ständig nacheinander alle Tasten ab (»scannt« sie) und meldet im Falle einer gedrückten Taste deren Code an das TOS. Dieser Scan-Code hat mit ASCII überhaupt nichts zu tun. Das TOS nun wandelt den Scan-Code in den ASCII-Code, dies aber nur für normale ASCII-Tasten, z. B. die Buchstaben. Andere Tasten werden nicht umgesetzt, dazu gehören u.a. die Funktionstasten. Die GEMDOS-Funktion CONIN meldet aber beide Codes in D0 zurück. Nenne ich den Scan-Code SC und den ASII-Code AC, so steht im Langwort D0 (also in 4 Bytes):

Byte 3	Byte 2	Byte 1	Byte 0
00	SC	00	AC

Anders ausgedrückt: Der Scan-Code steht im höherwertigen Wort, der ASCII-Code im niederwertigen. Im Falle von Nicht-ASCII-Codes jedoch findet man nur den Scan-Code, der ASCII-Code ist dann Null. Das ist auch der Grund, warum in vielen Hochsprachen der Scan-Code nicht zugänglich ist.

Wenn Sie nun das Programm lt. Bild 5.1 laufen lassen und dabei die Funktions-Tasten betätigen, werden Sie feststellen:

Taste	hat den Code (in hex)
F1	003B
F2	003C
F3	003D
usw. bis	
F10	0044

5.4 Lesen der Funktions-Tasten

Dies wissend, wollen wir ein kleines Programm schreiben, das für die Tasten F1 bis F9 die Zeichen 1 bis 9 ausgibt, bei F10 soll das Programm enden. Die Lösung zeigt Bild 5.2

```
* Programm f1
start
        move.w    #1,-(sp)      ; Funktion CONIN
        trap      #1            ; aufrufen
        addq.l    #2,sp
        swap      d0            ; Scan-Code nach d0.w
        cmpi      #$44,d0       ; Taste F10
        beq       fini          ; wenn ja, fertig
        subi      #$3B,d0       ; wandle in 0..9
        add       #49,d0        ; und nun in '1' bis
        move      d0,-(sp)      ; wollen wir ausgeben
        move      #2,-(sp)      ; Funktion CONOUT
        trap      #1
        addq.l #4,sp
        bra       start
fini    clr       -(sp)
        trap      #1
        end
```

Bild 5.2: Anzeige der Funktions-Tasten

Wir lesen (wie schon oft geübt) ein Zeichen mittels CONIN. Jetzt kommt ein neuer Befehl, nämlich

```
        swap    d0
```

»to swap« heißt vertauschen und getauscht werden hier die beiden Worte, aus denen ein Langwort besteht. Sie sehen, der 68000 bietet für jedes Problem den passenden Befehl. Nach dem Swap steht der Scan-Code im unteren Wort, womit wir ihn viel einfacher handhaben können. Zuerst testen wir, ob es die Taste F10 ist (#$44), wenn ja, sind wir fertig. Nun folgt etwas Mathematik. Vom Scan-Code subtrahieren wir $3B und addieren #49 (ASCII-Null plus 1). Heraus kommt 48 bis 57, sprich, der ASCII-Code für 1 bis 9, und den können wir mittels CONOUT ausgeben.

5.5 Die Mehrfachverzweigung

Nun ist es ja üblich, daß auf den Druck einer Funktions-Taste hin ein Programm etwas mehr tut, sprich, eine Routine aufruft. Das können sehr komplizierte Programmteile sein, wir wollen aber das Prinzip üben in der Art von

```
        IF  F1    THEN  GOTO  ...
        IF  F2    THEN  GOTO  ....
        u.s.w.
```

Die aufgerufene Routine soll auch nur 1, 2 usw. ausgeben, und damit nicht soviel zu tippen ist, schon bei F3 enden. Bild 5.3 zeigt die Lösung.

```
* Programm f2

start
        move.w  #1,-(sp)      ; Funktion CONIN
        trap    #1            ; aufrufen
        addq.l  #2,sp
        swap d0               ; Scan-Code nach d0.w

        cmpi    #$3B,d0       ; Taste F1?
        beq     f1            ; Wenn ja
        cmpi    #$3C,d0       ; Taste F2?
        beq     f2            ; wenn ja
        cmpi    #$3D,d0       ; Taste F3?
        beq      fini         ; wenn ja
        bra      start        ; andere sind nicht erlaubt

f1      move    #'1',-(sp)    ; wollen wir ausgeben
        move    #2,-(sp)      ; Funktion CONOUT
        trap    #1
        addq.l  #4,sp
        bra      start
```

```
f2      move    #'2',-(sp)   ; wollen wir ausgeben
        move    #2,-(sp)     ; Funktion CONOUT
        trap    #1
        addq.l  #4,sp
        bra       start

fini    clr       -(sp)
        trap    #1

        end
```

Bild 5.3: Einfachste Lösung der Mehrfach-Verzweigung

5.5.1 Lösung 1: Mit vielen »IF THEN«

Ich glaube, das Programm muß ich nicht mehr groß erklären, aber jedes Programm hat einen Sinn. Notfalls kann es nämlich als abschreckendes Beispiel dienen, und das soll es auch. Stellen Sie sich vor, ein Programm mit 100 Kommandos oder mehr würde so arbeiten. Das wäre ein Umstand und ein Spaghetti-Code!!

5.5.2 Lösung 2: ON X GOSUB in Assembler

Viel eleganter löst man so etwas mit einem »ON X GOSUB«. Dazu müssen wir zwar wieder einmal den Schwierigkeitsgrad etwas steigern, aber wenn Sie das gelernt haben, ist eigentlich schon fast alles gelaufen. Jedes Programm enthält nämlich eine Hauptschleife, in der es auf Kommandos wartet. Die Kommandos werden interpretiert und die passenden Unterprogramme aufgerufen. Danach geht es weiter in der Hauptschleife. In Basic sähe das so aus:

```
10    INPUT KOMMANDO
20    ON KOMMANDO GOSUB   100,200,300, ....
30    GOTO 10
```

Wie das in Assembler aussieht, zeigt Bild 5.4

```
* Programm f3
start
        move.w  #1,-(sp)     ; Funktion CONIN
        trap    #1           ; aufrufen
        addq.l  #2,sp
        swap     d0          ; Scan-Code nach d0.w

        subi    #$3B,d0      ; in 0..10 wandeln
        asl     #2,d0        ; mal 4
```

```
        lea       table,a0      ; Zeiger auf Tabelle
        move.l    0(a0,d0),a0
        jsr       (a0)
        bra       start

f1      move      #'1',-(sp)    ; wollen wir ausgeben
        move      #2,-(sp)      ; Funktion CONOUT
        trap      #1
        addq.l    #4,sp
        rts

f2      move      #'2',-(sp)    ; wollen wir ausgeben
        move      #2,-(sp)      ; Funktion CONOUT
        trap      #1
        addq.l    #4,sp
        rts

fini    move.l    (sp)+,d0      ; kill return address
        clr       -(sp)
        trap      #1

table                           ; Adress-Tabelle
        dc.l      f1
        dc.l      f2
        dc.l      fini
```

Bild 5.4: ON X GOSUB in Assembler

Den Anfang des Listings kennen Sie schon: Der Scan-Code der Funktions-Tasten wird in die Zahlen 0 bis 9 gewandelt, so steht er dann im Register D0. Der Einfachheit halber bearbeiten wir auch hier nur die Funktions-Tasten F1, F2 und F3, also 0, 1 und 2 in D0. Bitte beachten Sie, daß alle Tests fehlen (es geht um das Prinzip!), bei Betätigung aller anderen Tasten also das Programm abstürzt. Wir wollen ein »ON D0 GOSUB« realisieren und brauchen dazu natürlich zuerst 3 Unterprogramme, die hier f1, f2 und f3 heißen. Diese Unterprogramme sind trivial. Sie geben nur mittels CONOUT die Zeichen 1, 2 bzw. 3 aus. Nun betrachten wir das Listing wieder von unten. Da steht die Marke »table« und das, genau das, ist das Geheimnis unseres »ON X GOSUB«.

Für Mehrfachverzweigungen braucht man in Assembler

eine Sprungtabelle

Diese Tabelle ist eine Liste mit den Adressen der Unterprogramme.

Das Erstellen der Tabelle ist recht einfach. Schreiben Sie für jede Adresse

```
dc.l   Label
```

wobei für Label die Marke (die symbolische Adresse) des jeweiligen Unterprogramms einzutragen ist. Wichtig ist die Reihenfolge. Wir haben hier die Zuordnung

Taste	Routine
F1	f1
F2	f2
F3	fini

Die Reihenfolge der Tasten muß sich in der Reihenfolge der Einträge (so nennt man das) der Tabelle wiederholen. Die Unterprogramme selbst können in beliebiger Reihenfolge im Listing stehen. Es besteht also eine enge Zuordnung zwischen den Kommandos (hier den Funktionstasten) und der Sprungtabelle. Deshalb kann man aus den Kommandos die zugehörigen Adressen berechnen. Eine Adresse belegt immer 4 Bytes, also kann unsere Tabelle zum Beispiel so im Speicher stehen

Label	Adresse
f1	1000
f2	1004
fini	1008

Unser Kommando (in D0) kann sein

Wert	0,	1	oder	2
das mal 4 ergibt	0,	4	oder	8
und das plus 1000	1000,	1004	oder	1008

So einfach ist das also. Multiplizieren wir nun D0 mit 4. Dafür hat der 68000 natürlich auch einen speziellen Befehl (MULU), aber genau den nehmen wir hier nicht. Ein anständiger Assembler-Programmierer wird nämlich bei einer Multiplikation mit 2 oder 4 oder 8 oder 16 (Sie merken es: bei jeder 2er-Potenz) sofort hellhörig und greift auf einen Befehl zu, der das viel schneller erledigt. Hier heißt dieser Befehl

```
ASL   Arithmetic Shift Left
```

»ASL #1,d0« zum Beispiel schiebt alle Bits in D0 um 1 Stelle nach links. Die Wirkung ist die gleiche wie die im Dezimalsystem, wo Sie durch Linksschieben der Zahlen (und Festhalten des Kommas) mit 10 multiplizieren. Hier sind wir aber im dualen Zahlensystem, womit sich nur eine Multiplikation mit 2 ergibt. Schieben wir aber um 2 Stellen, so hätten wir schon unser »mal 4«. Das Ergebnis müssen wir auf den Beginn der Tabelle addieren. Deren Startadresse beschaffen wir uns mit »lea table,a0«. Nun kommt ein ganz wilder Befehl, nämlich

```
move.l 0  (a0,d0),a0
```

Wir benutzen die Adressierungsart »ARI mit Index und Offset«. Nur »Index« gibt es leider nicht, also setzen wir das Offset zu Null. Demnach errechnet sich die effektive Adresse als die Summe von A0 und D0. Die müssen wir nun in das Zielregister »moven«, und da nehmen wir gleich wieder A0. So etwas ist beim 68000 erlaubt, und weil es schön »tricky« aussieht, schreibt es auch jeder so. A0 zeigt nun also auf die Adresse des zugehörigen Unterprogramms, und das können wir nun schlicht mit »jsr (a0)« aufrufen. Nach dem »jsr« kehrt das Programm zu dem Befehl zurück, der dem »jsr« folgt. Der heißt »bra start«, also wieder von vorn mit dem nächsten Kommando.

Die Ausnahme von dieser Regel finden Sie im Unterprogramm »fini«. Dieses Unterprogramm ist gar keines, es wird zwar mit JSR aufgerufen, es endet aber nicht mit RTS. Folglich müssen wir die noch auf dem Stack befindliche Return-Adresse entfernen. Die Amerikaner sagen dafür so schön »Kill Return Address«. Wir erledigen das hier mit dem Befehl »move.l (sp)+,d0«. Das ist zulässig, weil in diesem Fall D0 nicht mehr benötigt wird. Es hat sich aber auch die folgende Schreibweise eingebürgert:

```
move.l    (sp)+,(sp)
```

Damit wird die Return-Adresse vom Stack geholt und gleich wieder auf den Stack geschrieben. Wegen des »+« hat sich auch der Stackpointer geändert, der Zweck ist also erreicht. Natürlich wäre auch ein »addq.l #4,sp« korrekt, aber einige Leute haben so ihre speziellen Methoden, um eine Aktion wie »Kill Return Address« herauszustellen.

5.5.3 Lösung 3: CASE X OF

Ist doch ganz einfach, oder? Es ist Ihnen zu einfach? Nun gut, machen wir die Sache etwas komplizierter. Unser schöner Kommando-Interpreter hat einen Nachteil. Die Kommandos müssen in der Reihenfolge von Scan-Codes auftreten. Auch andere Folgen, wie 1 bis 9 oder A bis M, sind denkbar, es muß aber immer eine Folge sein. Sie wissen jetzt, warum manche Leute ein Menü anbieten der Art

1 = Eingabe
2 = Rechnen
3 = Sichern

Das merkt sich schlecht, besser wäre doch

E = Eingabe
R = Rechnen
S = Sichern

5.6 Arbeiten mit zwei Tabellen

Das Prinzip ist natürlich wieder ganz einfach. Es gibt 2 Tabellen. In der ersten Tabelle stehen die »Keys« (die erlaubten Tasten oder Kommandos), in der zweiten die Adressen der zugehörigen Routinen. So muß man doch nur den Key in der ersten Tabelle suchen und aus seiner Platznummer in dieser Tabelle einen Zeiger auf den richtigen Platz in der Adreßtabelle errechnen. Da kann man dann die Adresse herausholen und ab geht's.

Diesmal soll unser Programm aber wasserdicht sein. Folglich muß der Fall »nicht gefunden« abgefangen werden. Wir wollen auch den User nicht zwingen, immer die Shift-Taste zu betätigen. Groß- und Kleinbuchstaben sollen deshalb gleich behandelt werden. Schließlich soll das Programm universell sein, sprich: es muß mit minimalem Aufwand möglich sein, Funktionen hinzuzunehmen oder zu ändern.

Nun denn, hier ist Bild 5.5 mit dem Listing:

```
* Programm menu

start
        move     #7,-(sp)        ; conin ohne Echo
        trap     #1
        addq.l   #2,sp
        tst       d0             ; kein ASCII-Zeichen?
        beq       start          ; wenn so
        cmpi     #'A',d0         ; kein Buchstabe?
        blt       start          ; dann ignoriere Taste
        bclr     #5,d0           ; Erzwinge Grossbuchstaben

* Suche Tasten-Code in Tabelle
* ----------------------------
        lea       keys,a0        ; Tabelle gueltige Keys
        move     #count,d1       ; deren Anzahl
search  cmp.b     (a0)+,d0       ; Key auf aktuellem Platz?
        dbeq      d1,search      ; wenn nicht, weitersuchen
                                   bis Tabellenende
        tst       d1             ; Key gefunden?
        bmi       start          ; wenn nicht, auf ein Neues

* Suche Adresse zu Key
* --------------------
        neg       d1             ; sub d1,#count
        add      #count,d1       ; ergibt Platznr. von Key
        lsl      #2,d1           ; die mal 4
        lea       table,a0       ; Adr. der Routine
        move.l    0(a0,d1.w),a0  ; bestimmen
        jsr       (a0)           ; und diese aufrufen
        bra       start          ; u.s.w.

Eingabe move     #'E',-(sp)
        move     #2,-(sp)
        trap     #1
        addq.l   #4,sp
        rts

Rechnen move     #'R',-(sp)
        move     #2,-(sp)
        trap     #1
        addq.l   #4,sp
        rts

Sichern clr       -(sp)          ; Fall Exit
        trap     #1
```

```
        data
keys    dc.b    'E','R','S'
count   equ     *-keys

        ds.w    0
table   dc.l    Eingabe
        dc.l    Rechnen
        dc.l    Sichern

        end
```

Bild 5.5: Menü-Technik universell

Diesmal beginne ich am Anfang. Wir lesen wieder eine Taste, jetzt aber nicht mit der GEMDOS-Funktion #1, sondern mit #7. Der einzige Unterschied ist, daß hier das Zeichen, das sie eintippen, nicht auf dem Bildschirm erscheint. Man nennt das »ohne Echo«. In D0 steht danach der Tasten-Code wie gehabt, und den teste ich nun. Um einen Operanden zu testen, gibt es den Befehl

```
        tst     Operand
```

Der TST-Befehl prüft nur, ob der Operand negativ ist (oder nicht) und ob er Null ist (oder nicht). Entsprechend werden die Flags N und Z gesetzt. Ich will nur verhindern, daß das Programm an dieser Stelle auf Sondertasten reagiert. Diese haben keinen ASCII-Code, folglich ist D0 – genau: das untere Wort in D0 – auf Null abfragen. Bei ungültiger Eingabe geht es mit »beq start« zum Warten auf die nächste Taste. Ich will aber auch alle Zeichen, die im ASCII-Code unter A stehen (<65) verbieten. Deshalb folgt dem »cmpi #A,d0« ein »blt start« (branch if less than = springe, wenn kleiner als). Nun will ich noch einen evtl. Kleinbuchstaben in einen Großbuchstaben wandeln und das geschieht mit einem Trick, den so direkt nur Assembler-Programmierer anwenden können.

Force Uppercase

Force Uppercase nennen das die Fachleute in Neudeutsch. Wenn Sie einmal auf eine ASCII-Tabelle schauen (siehe Anhang), wird Ihnen sicherlich auffallen, daß sich die Klein- und die Großbuchstaben immer um 32 unterscheiden. 2 hoch 5 ist aber auch 32, sprich, im Falle von Kleinbuchstaben ist Bit 5 gesetzt. Folglich wird ein guter (sind Sie doch) Assembler-Programmierer nicht sagen: »wenn der Code > Z ist, dann subtrahiere 32«, sondern er sagt einfach »lösche Bit 5«. Auch dafür kennt der 68000 einen Befehl, nämlich »BCLR« (Bit Clear).

```
        bclr    #5,d0
```

löscht Bit 5 (setzt es auf 0) im Operanden (hier D0).

An dieser Stelle sind wir also soweit, daß wir prüfen können, ob der User E, R oder S eingegeben hat. Wenn Sie nun bitte einmal auf das Ende des Listings schauen: Da gibt es eine kleine Tabelle für diese 3 Zeichen.

5.7 Der Location-Counter und Equates

Darunter aber steht

```
count  equu  *-keys
```

Wow, das ist ein Ding! Fangen wir mit »equ« an. »equ« ist ein sogenanntes Equate (englisch), Sie lesen es aber am besten als »equal = gleich.

Die Assembler-Direktive von zum Beispiel

```
GEMDOS   equ  1
```

bedeutet, ab jetzt kann man anstatt 1 auch GEMDOS sagen. »trap #GEMDOS« wäre dann ein gültiger Befehl. Prinzipiell ist das nichts weiter als reine Textverarbeitung. Der Assembler setzt einfach nachher für GEMDOS immer eine 1 ein. Wir können GEMDOS aber jetzt auch als Konstante ansehen. Falls Sie Pascal können, betrachten Sie das »equ« wie »const«, als C-Programmierer wie ein »#define«.

Das nächste »Wow« ist der »*«. Als erstes Zeichen in einer Programmzeile ist er ganz harmlos und bedeutet nur »Kommentar folgt«. Als Operand heißt er »Location Counter (LC)«. Sie wissen, daß Befehle, je nach Anzahl und Art der Operanden, verschieden viel Speicher belegen. Deshalb führt der Assembler einen Zähler, der sozusagen die bis zu jedem Befehl verbrauchten Bytes mitzählt. In diesem Sinne entspricht der LC dem PC (Programm Counter), mit dem nachher die CPU ein Programm verfolgt. Der Unterschied: Der LC wird auch durch Assembler-Direktiven inkrementiert, wie zum Beispiel »dc.b », das ja auch Bytes (mit Daten) belegt.

In der Assembler-Syntax heißt nun aber der LC nicht LC, sondern »*«. Betrachten wir ein Beispiel. Im Listing habe der LC zu Beginn der Zeile mit der Marke »keys« den Wert 100. Die Anweisung »dc.b E, R, S » läßt ihn auf 100 (für E), stellt ihn dann auf 101 (für R) und schließlich auf 102 (für S). Die nächste Zeile mit der Marke »count« sieht den LC als 103. An dieser Stelle erfolgt aber ein Equate. Da in Equates auch Ausdrücke erlaubt sind, hat

```
                    count equ *-keys
die Wirkung von     count = LC - keys
Das in Zahlen ergibt count = 103-100
```

Damit hätten wir unsere symbolische Konstante 3. Warum habe ich da nicht einfach »count equ 3« geschrieben? Antwort: Das machen nur Anfänger! Wenn wir nämlich später die Tabelle erweitern, können wir das tun, ohne die Zeile mit dem »equ« anfassen zu müssen. In jedem Assemblerlauf (der muß dann ohnehin sein) wird automatisch die richtige Anzahl eingesetzt. Außerdem: Tabellen können ganz schön lang sein, da kann man sich leicht verzählen.

5.8 Suchen mit DBcc

Nun müssen wir den Key (steht immer noch in D0) in der Tabelle »keys« suchen. Diesen Teil finden Sie unter »Suche Tasten-Code in Tabelle«. Das Problem: Auch »nicht gefunden« muß gemeldet, erkannt werden. Die Lösung ist eine DBcc-Schleife. Führen Sie sich bitte immer vor Augen

```
DBcc    dn,loop
```

heißt: verlasse die Schleife, wenn die Bedingung cc erfüllt ist, oder Dn auf -1 gelaufen ist, sonst springe zu »loop«.

Vorab wird mit »lea keys,a0« ein Zeiger auf den Beginn der Tabelle gestellt. Der Trick: Der Schleifenzähler D1 wird mit Count geladen (hier 3), eine DBcc-Schleife läuft aber im Grenzfall bis -1, hier also über 4 Schritte. D.h., wenn die Schleife wegen des »eq« (»equal« bedeutet gefunden) im »dbeq« verlassen wird, kann D1 nicht negativ sein. Ist es negativ, kommt das »bmi start« zur Wirkung (springe, wenn negativ).

Unter »Suche Adresse zu Key« taucht nun das nächste Problem auf. In der DBcc-Schleife lief D1 ja rückwärts. Es hat also diese Werte im Falle von »gefunden« :

Key	D1
1	3
2	2
3	1

Ich brauche aber die Folge 0, 1, 2, um wieder auf die Adreßtabelle wie im vorigen Beispiel zugreifen zu können. Dieses ergäbe sich ganz einfach, wenn ich von Count D1 subtrahieren würde. (3-3=0, 3-2=1, 3-1=2). Leider erlaubt der 68000 den Befehl »sub d1,#count« nicht; wie soll er auch von einer Konstanten etwas subtrahieren?

Da hilft die Anweisung

```
neg  d1
```

Ich negiere D1. War es 3, dann wird es -3. Darauf addiere ich Count, ergibt 0. Alte Regel: Wenn man nicht subtrahieren kann, muß man eben den negativen Wert addieren. Den Rest hatten wir schon. Als kleinen Unterschied habe ich hier nun anstatt »asl« »lsl« (logical shift left) genommen. Der Unterschied: ASL schiebt arithmetisch korrekt, würde also auch das Vorzeichen berücksichtigen, wenn wir eines hätten. Damit wären wir wieder an der Stelle angekommen, die das vorige Programm ausmachte. Mit »lea table,a0« zeigen wir auf die Adreßtabelle, und den Rest kennen Sie schon. Die Unterprogramme zeigen diesmal Buchstaben an, und »S« beendet die Geschichte und auch dieses Kapitel.

Kapitel 6

Das Betriebssystem im Detail

GEMDOS

BIOS

XBIOS

GEM

VDI

AES

In diesem Kapitel soll das Betriebssystem des Atari ST insgesamt vorgestellt werden. Schließlich ist es die Basis für jede Assembler-Programmierung. Bevor Sie auch nur eine Routine neu erfinden, sollten Sie immer hier nachsehen, ob es sie nicht schon gibt.

6.1 Wer macht was?

Über das BS – so nennt man ein Betriebssystem – des Atari konnte man schon einiges lesen, leider aber auch viel Falsches. Ursache dürfte u.a. sein, daß Atari in der Anfangsphase einige Zwischenstände hat durchsickern lassen, die dann nie zur Auslieferung kamen. Um nun evtl. noch kursierende Gerüchte (was hält sich länger?) auszuräumen, zuerst einige Klarstellungen. Das BS des ST hat mit CP/M 68K herzlich wenig gemeinsam. Wenn man schon Ähnlichkeiten mit anderen BS sucht, dann findet man ein wenig UNIX und sehr viel Übereinstimmung mit MS-DOS. Das Verwirrspiel um CP/M-68K lag daran, daß dies ursprünglich für den ST geplant war. Es mußte dann aber verworfen werden, weil es für ST-typische Aufgaben zu langsam ist.

Folglich schrieben die Atari-Leute flugs ihr eigenes BS (englisch OS [Operating System]) und nannten dieses nach ihrem Boß (Mr. Tramiel) TOS. TOS besteht aus

GEMDOS
BIOS,
XBIOS und einem Grafik-Kern.

GEMDOS hat noch nichts mit GEM zu tun, der Name täuscht! Es steht in der Hierarchie ganz oben. Es ist dem Benutzer am nächsten. Über dem TOS steht nur noch die sog. Benutzeroberfläche. Näheres dazu finden Sie im Absatz 6.5. Die unteren Ebenen des TOS sind das BIOS und das XBIOS.

BIOS heißt »Basic Input Output System« und stellt die Basis-Routinen für die Ein/Ausgabe zur Verfügung. So ein BIOS findet man in praktisch jedem BS, nur nicht jedes BS hat sich auch um solche Dinge wie eine Maus zu kümmern. Deshalb gibt es beim ST noch das XBIOS. XBIOS steht für extended (erweitertes) BIOS. Sinn dieser Trennung ist es, Programme – auch wenn sie in Assembler geschrieben sind – unabhängig von der Hardware zu machen. Auf dieser Ebene sagt man zum Beispiel nur »Zeichen ausgeben«. Das »wie« überläßt man dem BIOS. Folglich muß man bei Änderungen der Hardware (hier ein anderer Bildschirm) nur das BIOS ändern. Das hat aber zur Folge, daß jeder GEMDOS-Aufruf praktisch einen oder mehrere BIOS-Aufrufe bewirkt.

Nun können Sie mit Recht sagen, die Hardware meines ST ändert sich nicht, ich rufe das BIOS selbst auf. Dagegen ist überhaupt nichts einzuwenden. In diesem Buch folgen noch Programme, die genau das tun. Vorteil der Aktion ist, daß solche Programme zwangsläufig schneller sind, denn es entfällt eine Instanz.

Neben dem BIOS/XBIOS gibt es noch eine Sammlung von Grafik-Kernroutinen, die speziell das GEM unterstützen. Mit Hilfe dieser Routinen ist aber auch sehr gut Grafik ohne GEM möglich, diese ist dann sogar wesentlich schneller. Im Kapitel 9 wird diese Grafik ausführlich vorgestellt. GEMDOS, BIOS und XBIOS. sowie die Grafik und sogar das GEM werden immer über Trap-Befehle aufgerufen.

Dabei gilt:

Trap #1: GEMDOS
Trap #2: GEM
Trap #13: BIOS
Trap #14: XBIOS
A-Traps: Grafik

Für die Traps 1, 2, 13, und 14 ergibt sich immer diese Aufrufsequenz:

Parameter auf den Stack
Funktions-Nummer auf den Stack
Trap #1 (#2, #13, #14)
Stackpointer wiederherstellen (wie vor Aufruf)

Wenn ein Parameter länger als ein Wort ist, wird ein Zeiger auf ihn übergeben. Wir kennen das schon von den Strings her, wo wir immer eine Adresse übergeben haben. Die Rückgabe von Werten erfolgt im Register D0, das direkt den Wert oder einen Zeiger auf eine Tabelle mit Werten enthalten kann. Manchmal hat auch das Register A0 diese Zeigerfunktion. Die nun folgenden Tabellen sollen Ihnen nur einen Überblick über das doch sehr breite Angebot geben. Wie die einzelnen Funktionen aufgerufen und genutzt werden, wird ausführlich im Anhang geschildert.

6.2 GEMDOS-Funktionen (Trap #1)

Nr.	Name	Bedeutung
0	TERM	Return (zum Desktop oder Caller)
1	CONIN	Konsoleingabe
2	CONOUT	Konsolausgabe
3	AUXIN	Eingabe von RS-232
4	AUXOUT	Ausgabe auf RS-232
5	PRTOUT	Druckerausgabe (Par.-Port)
6	CONIO	Mit $FF Tastaturstatus, sonst wie 1
7	DIRCON	Wie 2 ohne Echo
8	DIRCON	Wie 7
9	PRTSTR	String ausgeben
10	RDSTR	String einlesen
11	CONSTAT	Tastaturstatus prüfen
14	SETDRV	aktuelles Laufwerk setzen
16	SRNSTAT	Status Schirm prüfen
17	PRTSTAT	Drucker-Status
18	AUXINSTAT	Zeichen von RS-232 verfügbar?
19	AUXOUTSTAT	RS-234 bereit?
25	CURDISK	aktuelles Laufwerk ermitteln
26	SETDTA	Adresse für DTA-Puffer setzen

32	SUPER	CPU in Supervisor-Mode
42	GETDATE	Datum holen
43	SETDATE	Datum stellen
44	GETTIME	Uhr lesen
45	SETTIME	Uhr stellen
47	GETDTA	DTA-Adresse holen
48	GETVER	TOS-Versionsnummer lesen
49	KEEPPROC	Return, aber gebe Speicher nicht frei
54	FRESPC	Freien Speicher auf Disk ermitteln
57	MKDIR	Sub-Directory einrichten
58	RMDIR	Sub-Directory löschen
59	CHDIR	Directory-Pfad setzen
60	CREATE	File anlegen
61	OPEN	File öffnen
62	CLOSE	File schließen
63	READ	Lese Anzahl Bytes von File
64	WRITE	Schreibe Anzahl Bytes auf File
65	UNLINK	File löschen
66	LSEEK	File-Pointer stellen
67	CHMOD	File-Attribute ändern
69	DUP	File-Handle zuweisen Linking)
70	FORCE	File-Rooting
71	GETDIR	Sub-Dir. zum aktuellen machen
72	MALLOC	Speicher reservieren
73	MFREE	Reservierten Speicher von 72 freigeben
74	SETBLCK	Definierten Speicherbereich reservieren
75	EXEC	Pgm nachladen und starten
76	TERM	Wie TERM 0, aber mit Rückgabe-Parameter
78	SFIRST	Pfad/File suchen
79	SNEXT	nächsten File (nach SFIRST) suchen
86	RENAME	Filenamen ändern
87	GSDTOF	File-Größe und Erstelldatum lesen

6.3 BIOS-Funktionen (Trap #13)

Nr.	Name	Bedeutung
0	GETMPB	Memory-Parameter-Block kreieren
1	BCONSTAT	Status eines Eingabegerätes holen
2	BCONIN	Zeichen von Gerät holen
3	BCONOUT	Zeichen an Gerät senden
4	RWABS	Sektor lesen oder schreiben
5	SETEXEP	Exception-Vektor setzen
6	TICKCAL	System-Timer lesen

7	GETBPB	BIOS-Parameter-Block lesen
8	BCOSSTAT	Status eines Gerätes holen
9	MEDIACH	Diskette gewechselt?
10	DRVMAP	Bit-Map angemeldeter Drives holen
11	KBSHIFT	Status der Tasten Shift, CRTL, ALT, Maus

6.4 XBIOS-Funktionen (Trap #14)

Nr.	Name	Bedeutung
0	INITMOUSE	Maus-Parameter setzen
1	SSBRK	Speicher reservieren (oben)
2	PHYSBASE	Physikal. Adresse des Video-RAM lesen
3	LOGBASE	Logische Vido-Adresse lesen
4	GETRES	aktuelle Video-Auflösung lesen
5	SETSCRN	Video-Parameter setzen
6	SETPALET	Farbpalette laden
7	SETCOLOR	Farbe setzen
8	FLOPRD	n Sektoren lesen
9	FLOPWR	n Sektoren schreiben
10	FLOPFMT	Track formatieren
12	MIDIWS	String an MIDI ausgeben
13	MFPINT	MFP-Interrupt initialisieren
14	IOREC	Zeiger auf IO-Buffer eines Gerätes holen
15	RSCONF	RS-232 konfigurieren
16	KEYTBL	Neue Tastaturtabellen anlegen
17	RANDOM	Zufallszahl erzeugen (24 Bit)
18	PROTOBT	Boot-Sektor erzeugen
19	FLOPVER	Sektoren mit Puffer verifizieren
20	SRNDUMP	Bildschirm-Hardcopy auf Drucker
21	CURSOR	Cursor-Parameter setzen
22	SETTIME	Datum und Zeit setzen
23	GETTIME	Datum und Zeit lesen
24	BIOSKEYS	Standard-Tastenbelegung (wieder) herstellen
25	IKBWS	Befehle an Keyboard-Kontroller geben
26	JDISINT	MFP-Interrupt sperren
27	JENABINT	MFP-Interrupt freigeben
28	GIACCESS	GI-Sound-IC-Register setzen
29	OFFGIBIT	Bit in Port des GI off (aus)
30	ONGIBIT	"-" on (an)
31	XBTIMER	Timer im MPF starten
32	DOSOUND	Soundstring abarbeiten
33	SETPRT	Druckerkonfiguration lesen/schreiben
34	KBDVBASE	Adr. Vektor-Tabelle f. KBD-Prozessor

35	KBRATE	Tastatur-Repeat setzen
36	PRTBLK	Video-Dump ab Adresse
37	WVBL	Auf Bildrücklauf warten
38	SUPEXEC	Routine im Supervisor-Mode ausführen
39	PUNTAES	AES abschalten (solange nicht im ROM)

6.5 GEM, VDI und AES

Bisher hatten wir das Betriebssystem – das TOS – des Atari ST betrachtet. Eine wichtige Eigenschaft fehlt jedoch noch, nämlich die Benutzeroberfläche. Das bedeutet schlicht: Wenn der Computer eingeschaltet wird, dann soll er sich melden in der Art »Bin bereit, warte auf Ihre Befehle, verehrter Herr und Meister«. Im einfachsten Fall zeigt er dazu einen sog. Prompt, beispielsweise das Zeichen ">". Etwas fachmännischer ausgedrückt, das System wartet auf ein Kommando. Das Kommando wird interpretiert und dann – falls erkannt – ausgeführt. Dieses tut natürlich auch ein Programm, nämlich der Kommando-Interpreter. Diese Art der Bedienung können Sie beim Atari ST auch haben, Sie benötigen dazu ein Programm namens COMMAND.PRG. Dieser Kommando-Interpreter ist Teil des Atari-Entwicklungspakets, ist aber auch als »public domain« aufzutreiben. Durch Änderung des Boot-Sektors der Startdiskette (siehe Kapitel 8) kann man sogar dafür sorgen, daß sich der ST nicht mit dem Desktop, sondern mit diesem Kommando-Interpreter meldet. Diese Benutzeroberfläche macht sich gerade für die Assembler-Programmierung oft recht gut.

Prinzipiell kann man den ST unter COMMAND.PRG genauso bedienen, wie einen IBM-PC, bei dem dieses Programm COMMAND.COM heißt. Wie auch immer, jeder Computer sorgt dafür, daß nach dem Einschalten oder einem Reset der Kommando-Interpreter von der Diskette geladen und gestartet wird bzw., wenn er im ROM ist, nur gestartet wird.

Der große Unterschied zwischem dem Atari und dem IBM-PC ist nun der, daß beim Atari ST nicht COMMAND.COM gestartet wird, sondern ein Programm namens GEM.

Der Atari startet immer unter TOS. Das TOS lädt dann das GEM oder aktiviert es, wenn im ROM ist. Das GEM ist also eine Benutzeroberfläche wie COMMAND.PRG. Der Unterschied ist nur, daß dieser Kommando-Interpreter hauptsächlich mit der Maus bedient wird. Ein paar Vergleiche:

Unter GEM	Unter COMMAND.PRG (tippen)
Neuen Ordner anlegen	md (Make Directory)
Dokument kopieren	copy alter_name neuer_name
Dokument in Papierkorb	delete name
Programm anklicken	Name eintippen

GEM heißt »Graphic Environment Manager«, also Manager der Grafik-Umgebung. Natürlich hat das GEM eine Menge zu tun, wie beispielsweise das Bereitstellen der Menüs und der Fenster sowie deren Überwachung.

Diese sehr weit oben angesiedelten Aufgaben des GEM werden von den AES, den »Application Environment Services«, also den Anwendungs-Umgebungs-Diensten, ausgeführt. Die AES (die Dienste) müßte man also sagen, es hat sich jedoch das AES eingebürgert. Das AES zeichnet nun zum Beispiel ein Fenster, indem es entsprechende Routinen des VDI aufruft. VDI heißt »Virtual Device Interface«, sprich, es ist ein virtuelles (scheinbares) Zeichengerät. Das VDI wiederum greift auf Grafik-Grundfunktionen zu wie zum Beispiel Zeichen einer Linie. Diese »Drawing Primitives« (am besten mit Grafik-Kernroutinen übersetzt), werden beim ST über den Line-A-Emulator, auch kurz A-Traps genannnt (siehe Kapitel 9), aufgerufen.

Der Aufruf von GEM-Routinen in Assembler erfolgt nach diesem Muster:

```
move.l    #paramterblock, d1
move.l    #magic,d0
trap      #2
```

"parameterblock" ist eine Sammlung von Arrays, über die die Parameterübergabe läuft. »magic« ist eine geheimnisvolle Zahl (Atari weiß warum) mit der Bedeutung:

```
$C8   bei  AES-Aufrufen
$73   bei  VDI-Aufrufen
```

Die normale Hierarchie wäre

```
        Anwender

        GEM

            Anwender-Programm

AES     VDI             GEMDOS

                        BIOS

        A-Traps         XBIOS
```

Als Assembler-Programmierer haben Sie die Freiheit, sich aus diesen immer vorhandenen Programm-Bibliotheken das jeweils Passende auszusuchen. AES und VDI sind jedoch derart umfangreich, daß ich hierzu auf spezielle Literatur verweisen muß.

Kapitel 7

Rationalisierung der Arbeit

Strukturierung von Assemblerprogrammen

Makros

Include-Files

Module

Programm-Entwicklung am Beispiel »bindec«

Wissen Sie eigentlich, »wieviel Programm« ein professioneller Programmierer pro Tag schreibt? 6 (in Worten sechs) Zeilen pro Tag! Diese Zahl ergibt sich, wenn man den Gesamtaufwand an Tagen, beginnend bei der Problemanalyse über die ersten Vorentwürfe, das eigentliche Programmieren, das Testen, das Debuggen, bis hin zur Dokumentation addiert und dann die Programmzeilen durch diese Anzahl Tage dividiert.

Dabei spielt die Programmiersprache keine Rolle. Es kommen immer diese sechs Zeilen dabei heraus.

Nun kann man natürlich mit 6 Zeilen Pascal viel mehr Wirkung erzielen als mit sechs Zeilen Assembler. Um so wichtiger ist es, gerade in der Assembler-Programmierung alle Möglichkeiten zur Rationalierung der Arbeit auszunutzen.

7.1 Strukturierung von Assembler-Programmen

Die wirkungsvollste Methode ist die Strukturierung der Programme. Damit ist primär nicht »WHILE...WEND« und ähnliches gemeint (ist auch vorteilhaft, siehe 7.1.1), sondern die grundsätzliche Struktur des Programms. Im allgemeinen sieht doch ein Programm so aus:

Logo
Menü
Warten auf Eingabe
Reaktion auf die Eingabe

Mit Logo ist das Bild gemeint, mit dem sich das Programm beim Start meldet. Dann werden dem Bediener die möglichen Funktionen des Programms in einem (Haupt-)Menü angeboten und auf eine Eingabe gewartet. Die Eingabe wird interpretiert und daraufhin die entsprechende Funktion aufgerufen. Im Ansatz hatten wir diese Technik im Kapitel 5 schon praktisch geübt. In Basic sähe das so aus:

```
100 PRINT "LOGO"
200 PRINT "MENU"
300 INPUT KEY
400 ON KEY GOSUB 1000,2000,3000,.....
410 GOTO 100
```

In Assembler ändert sich daran prinzipiell nichts. In der typischen Schreibweise dieser Sprache könnte man schreiben: (»bsr« heißt »Branch to Sub Routine«, also GOSUB)

```
        bsr  logo
loop    bsr  menu
        bsr  conin              ; Warte auf Eingabe
        bsr  rechne_adresse
        bsr  adresse            ; Funktion aufrufen
        bra  loop
;
; Beginn der Unterprogramme
```

Sie sehen, der wesentliche Ansatz ist die Gliederung in Unterprogramme. Diese Unterprogramme rufen selbst wieder Unterprogramme auf, und auch die »Unter-Unterprogramme« rufen ggf. nochmals Unterprogramme auf und seien es nur Funktionen des Betriebssystems. Gerade letztere zeigen jedoch einige wichtige Eigenschaften, die Unterprogramme haben sollten, nämlich

universelle Verwendbarkeit
und klar definierte,
einheitliche Schnittstelle.

Ein gutes Beispiel sind die Routinen des oben gezeigten Rahmens. Es ist ja durchaus möglich, daß Sie zu einigen Punkten des Hauptmenüs Untermenüs anbieten müssen. Dann ist es doch sehr praktisch, wenn Sie dafür dieselben Unterprogramme wiederum benutzen können.

7.1.1 Struktur in der Sprache

Im Kapitel 4 hatte ich Ihnen ein Beispiel vorgestellt, das die Buchstaben von A bis Z druckt. Hier noch einmal der wesentliche Teil des Programms:

```
        move    #25,d1      ; der Schleifenzaehler
        move    #'A',d2     ; Anfangszustand
loop    move    d2,-(sp)    ; Buchstabe auf Stack
        move    #2,-(sp)    ; Funktion CONOUT
        trap    #1          ; GEMDOS aufrufen
        addq.l  #4,sp       ; Stack korrigieren
        addq    #1,d2       ; naechster Buchstabe
        dbra    d1,loop
```

Bild 7.1: Drucken von A bis Z diskret geschrieben

Was halten Sie nun von der folgenden Alternative? (Ich garantiere, wir sind immer noch in Assembler)

```
        for d2 = #'A' to #'Z' do
          conout d2
        endfor
; oder:
        move #'A',d2
        while d2 le #'Z' do
```

```
        conout d2
        addq #1,d2
      endwhile
```

Bild 7.2: Zwei Lösungen mittels Makros

Bild 7.2 ist nur ein kleiner Auszug aus der Makro-Sprache des GST-Assemblers. Wenn ich Ihnen jetzt noch sage, daß der daraus entstehende Code der gleiche ist, wie der aus der diskreten Lösung von Bild 7.1, dann wird Sie das Thema sicherlich interessieren.

7.2 Makros

Makro ist die Kurzform von Makro-Befehl. Makro selbst heißt so viel wie groß. Prinzipiell ist ein Makro nur eine Zusammenfassung einer Gruppe von Einzelbefehlen, wie wir sie bisher kannten, unter einem neuen Namen. Man kann die Einzelbefehle auch Mikro-Befehle nennen. Leider ist die Makro-Sprache nicht genormt. Jeder Assembler hat da seine eigene Syntax, die des eben gezeigten GST-Assemblers ist sogar ziemlich ausgefallen. Ich bringe deshalb alle folgenden Beispiele in der Makrosprache des Metacomco-Assemblers, die den »Makro-Dialekten« der meisten Assembler am ehesten entspricht.

Hier ein Beispiel:

```
conin   macro
        move.w  #1,-(sp)
        trap    #1
        addq.l  #2,sp
        endm
```

Dieser Makro realisiert die Funktion CONIN, wie wir sie schon kennen. Jeder Makro hat einen Namen, der im Label-Feld stehen muß, gefolgt von dem Schlüsselwort »macro«. Ein Makro endet mit dem Schlüsselwort »endm«. Zwischen »macro« und »endm« kann eine beliebige Anzahl von Befehlen stehen. Ist der Makro einmal definiert, kann er beliebig oft mit seinem Namen aufgerufen werden. Innerhalb eines Makros dürfen auch die Namen anderer, dann schon vorher definierter Makros, stehen. Ob und wie tief Makros so geschachtelt werden dürfen, lesen sie aber besser in Ihrem Handbuch nach.

Bild 7.3 bringt drei Makros, so wie Sie sie einfach zu Beginn eines Programms tippen können.

```
writes  macro
        pea     \1              ; "\" muß ein Backslash sein!
        move.w  #9,-(sp)
        trap    #1
```

```
        addq.l    #6,sp
        endm
conin   macro
        move.w    #1,-(sp)
        trap      #1
        addq.l    #2,sp
        endm
term    macro
        clr       -(sp)
        trap      #1
        endm
```

Bild 7.3: Drei Makros, die man immer braucht

Der Makro »writes« wie »write string« beinhaltet die Ihnen schon bekannte GEMDOS-Funktion #9. Neu ist hier, daß wir dem Makro einen Parameter übergeben müssen, nämlich den auszugebenden String. Für solche Parameter haben Makros Variable. Bei GST dürfen dies Namen sein, meistens üblich sind aber Ziffern mit einem Schrägstrich oder Fragezeichen davor. Beachten Sie bitte, daß der Schrägstrich bei Metacomco ein »Backslash« (nach links gekippter Strich) sein muß. Solche Einschränkungen sind zwar unschön (das Zeichen gibt es nicht in jedem Editor), aber übliche Unsitte. Die beiden anderen Makros haben Sie sicherlich schon erkannt, so daß wir uns jetzt der Anwendung widmen können.

Das Programm soll lediglich die beiden Strings »msg1« und »msg2« ausgeben und dann auf eine Taste warten. Bild 7.4 zeigt die Lösung als Fortsetzung von Bild 7.3.

```
        writes    msg1
        writes    msg2
        conin
        term
msg1    dc.b      'Hallo, ',0
        ds.w      0
msg2    dc.b      'Atari',0
        end
```

Bild 7.4: Ein Programm mit Makros

Das sieht doch richtig gut aus. Was ist nun der Haken an der Sache? Dazu möchte ich Ihnen mit Bild 7.5 ein sogenanntes Assembler-Listing vorstellen.

```
MC68000 ASSEMBLER VERSION 10.193

   LOC                  OBJECT   STMT              SOURCE
STATEMENT

                                  2          pea     \1
                                  3          move.w  #9,-(sp)
                                  4          trap    #1
                                  5          addq.l  #6,sp
                                  6          endm
                                  7
                                  8   conin  macro
                                  9          move.w  #1,-(sp)
                                  10         trap    #1
                                  11         addq.l  #2,sp
                                  12         endm
                                  13
                                  14  term   macro
                                  15         clr     -(sp)
                                  16         trap    #1
                                  17         endm
                                  18
                                  19         writes  msg1
0000'   4879   0000   0028        19+        pea     msg1
0006'   3F3C   0009               19+        move.w  #9,-(sp)
000A'   4E41                      19+        trap    #1
000C'   5C8F                      19+        addq.l  #6,sp
                                  20         writes  msg2
000E'   4879   0000   0030        20+        pea     msg2
0014'   3F3C   0009               20+        move.w  #9,-(sp)
0018'   4E41                      20+        trap    #1
001A'   5C8F                      20+        addq.l  #6,sp
                                  21         conin
001C'   3F3C   0001               21+        move.w  #1,-(sp)
0020'   4E41                      21+        trap    #1
0022'   548F                      21+        addq.l  #2,sp
                                  22         term
0024'   4267                      22+        clr     -(sp)
0026'   4E41                      22+        trap    #1
0028'   4861   6C6C   6F2C 2000   24  msg1   dc.b    'Hallo, ',0
0030'   =0000                     25         ds.w    0
```

```
0030'   4174   6172   6900          26 msg2   dc.b    'Atari',0
                                    27
                                    28 end
No errors found in this Assembly
```

Bild 7.5 Ein Assembler-Listing

So ein Assembler-Listing erzeugen die meisten Assembler nur auf Wunsch. In der Regel muß man dafür die sog. List-Option einschalten. Die einzelnen Felder bedeuten:

LOC : Der Wert des »Location Counters«, siehe Kapitel 5.7.

OBJECT: Der Objekt-Code in hex, also die Maschinensprache. Sie können daran sehr gut ablesen, welche Anweisung wie übersetzt wird und wieviele Worte eine Anweisung belegt.

STMT: Einfach eine Zeilennummer zu Ihrer Orientierung.

SOURCE STATEMENT: Der Quelltext, den Sie eingetippt haben.

In den Zeilen 1-18 stehen die Makrodefinitionen. In Zeile 19 wird das erste Mal der Makro »writes« aufgerufen. Nun folgt viermal die Zeile »19+«. Das nennt man die Makroentwicklung. Sie sehen, der Assembler hat einfach die entsprechenden echten Anweisungen eingesetzt. Das nächste »writes« in Zeile 20 hat die gleiche Makroentwicklung zur Folge, nur daß jetzt anstatt »msg1« für den Platzhalter »msg2« eingesetzt wurde. Beachten Sie bitte den Unterschied im Feld »OBJECT. Für »msg1« gilt die Adresse $28, »msg2« beginnt bei Adresse $30. Das Wesentliche ist aber: Obwohl der Quelltext soviel kürzer aussieht, im Ergebnis sparen Sie nichts. Das ganze Makro-Prozessing ist reine Textverarbeitung!

Nun hätten wir noch ein kleines Problem. Nehmen wir an, Sie hätten einen Makro geschrieben der Art von

```
nonsens  macro
loop     move    d1,d2
         bra     loop
         endm
```

Schreiben Sie nun in Ihrem Programm

```
         nonsens
         nonsens
```

so wird der Makroprozessor daraus folgende Zeilen entwickeln:

```
loop     move    d1,d2
         bra     loop
loop     move    d1,d2
         bra     loop
```

Spätestens beim zweiten »bra loop« wird der arme Assembler ins Schleudern geraten. Zu welchem »loop« soll er denn nun springen? Praktisch wird er mit einer Fehlermeldung aussteigen. Um so etwas zu vermeiden, gibt es nun in guten Assemblern immer eine Lösung. Die sinnvollste Art findet man zum Beispiel bei GST, wo man schreiben würde

```
nonsens  macro
         LOCAL loop
loop     move  d1,d2
         bra   loop
         endm
```

Damit wird »loop« zur lokalen, also nur innerhalb des Makros gültigen Variablen erklärt. In anderen Assemblern findet man die Form

```
$n anstatt »loop«
```

Für n ist eine Zahl zwischen 1-99 (oder 1-999) einzusetzen. Diese Zahl wird bei jedem Aufruf des Makros um eins hochgezählt. Wenn Sie mehrere Makros mit internen Labels verwenden, müssen Sie allerdings darauf achten, daß Sie mit unterschiedlichen und ggf. weit auseinanderliegenden Zahlen starten. Um noch einmal auf das Beispiel aus der Einleitung zurückzukommen, nämlich

```
for d2 = #'A' to #'Z' do
  .....
  .....
endfor
```

so ist des Rätsels Lösung ganz einfach. »for«, »=«, »to« und »endfor« sind Makros. »do« beispielsweise wird nur eine lokale Marke erzeugen, »for« wird d2 laden und »endfor« ein »dbra d2,marke« generieren. Eine andere Anwendung wäre diese:

```
ret      makro
         rts
         endm
```

In diesem Fall wird der Makroprozessor immer nur für jedes »ret«, das in einem Text auftaucht, ein »rts« einsetzen. Sie können auf diese Art jeden Assemblerbefehl neu definieren. Nun kommt es aber noch schlimmer, nämlich hiermit:

```
ret      makro
         dc.b      $C9
         endm
```

»ret« heißt Return in Z80-Assembler.

Für die Z80-Maschinensprache muß aber ein »ret« in »$C9« übersetzt werden. Sie können somit auf Ihrem ST in 68000-Assembler ein Programm in Z80-Assembler schreiben. So etwas nennt man Cross-Assembler. Möglich machen dies Makros. Nun wissen Sie auch, wie man einen neuen Computer in Assembler programmiert, wenn es für das gute Stück noch gar keinen Assembler gibt.

7.3 Include-Files

Auch die Include-Anweisung kann man zuerst zusammen mit Makros sinnvoll einsetzen. Nehmen wir an, Sie haben die Makrodefinitionen von Bild 7.3 in einem Textfile mit dem Namen »Mac73« abgelegt. Dann können Sie das Programm von Bild 7.4 jetzt so schreiben, wie es Bild 7.6 zeigt.

```
        include "Mac73"

        writes  msg1
        writes  msg2
        conin
        term
msg1    dc.b    'Hallo, ',0
        ds.w    0
msg2    dc.b    'Atari',0
        end
```

Bild 7.6: Ein Programm mit Makros, die aus einem Include-File gelesen werden

Diese Methode ist sehr sinnvoll, denn Sie werden sicherlich in jedem Programm zahlreiche TOS-Funktionen einsetzen. Wenn Sie sich diese Funktionen einmal als Makros definiert und in Ihrem »Mac-File« abgelegt haben, ersparen Sie sich nicht nur eine Menge an Tipperei, sondern auch einiges an Zeit für die Fehlersuche wegen nicht gemachter Tippfehler. Es macht durchaus nichts, wenn im jeweiligen Programm viele der Makros nicht genutzt werden. Sie erzeugen ja dann auch keinen Code. Wird die »Mac-Lib« (Kürzel für Library = Bibliothek) zu groß, kostet es natürlich Zeit, wenn Sie der Assembler bei jedem Lauf einlesen muß. Auf einer RAM-Disk kann dies unter Umständen zu Platzproblemen führen. Deshalb sollten Sie Ihre »Lib« möglichst in mehrere kleine Files nach Sachgebieten aufteilen.

Im nächsten Kapitel werde ich noch ein Beispiel bringen, das mit sehr vielen Makros arbeitet und diesem die konventionelle Lösung gegenüberstellen. Sie werden sehen, daß man mit Makros ein Programm sehr gut strukturieren und nahezu als Klartext schreiben kann.

7.4 Module

Module sind ein weiteres Hilfsmittel, die Programmierarbeit zu rationalisieren. Die dahinter stehende Philosophie ist, daß man ein großes Programm in viele einzelne, voneinander möglichst unabhängige Blöcke oder Abschnitte, kurz Module genannt, unterteilt. Eine

Sprache wie Modula beispielsweise ist aus dieser Philosophie heraus geboren worden. In Assembler muß man zwei Arten von Modulen unterscheiden, nämlich

Textmodule
Code-Module

7.4.1 Textmodule

Textmodule hatten wir praktisch schon behandelt, es sind die Include-Files. Ein auf Textebene modulisiertes Programm, das den Arbeitstitel DED trägt, könnte beispielsweise so aussehen:

```
include   "gemdos.mac"
include   "bios.mac"
include   "ded_logo"
include   "ded_menu"
include   "ded_subs"
```

Vorteil der Textmodule ist zuerst, daß die Listings relativ kurz werden, wenn Sie immer einen Teil, der fertig geworden ist, als Textmodul ablegen. Wenn ich zum Beispiel bei der Entwicklung des Teils »ded_menu« bin, in dem natürlich ein Bug steckt, dann brauche ich mich nicht erst bis zur Zeile 477 vorzuarbeiten, sondern bin da schon bei Zeile 5.

Vorteil Nummer zwei wäre natürlich, daß man allgemein brauchbare Dinge, wie zum Beispiel die GEMDOS-Lib immer wieder verwenden kann.

7.4.2 Code-Module

Ein Nachteil der Text-Module ist, daß sie bei jedem Lauf neu assembliert werden müssen. Abhilfe bringen die Code-Module. Dazu werden einzelne Blöcke getrennt assembliert und nachher vom Linker mit dem Hauptprogramm zusammengebunden. Das klingt sehr gut, bringt aber zuerst einiges an Mehrarbeit mit sich und stellt auch einige Anforderungen an den Linker und die gesamte Programmierumgebung an sich. Um es gleich zu sagen, der Aufwand lohnt sich nur bei größeren bis sehr großen Programmen.

Beginnen wir mit der Mehrarbeit im Programm. In einem kompletten Programm können Sie beispielsweise ohne weiteres sagen

```
bsr print.
```

Steckt aber die Print-Routine in einem anderen Modul, so müssen Sie Ihrem Assembler mitteilen, daß »print« eine externe Routine ist. Gleiches gilt für Variablennamen. Aus diesen Gründen muß der Assembler Direktiven der Art

External,
Global
und/oder XREF

bieten. Diese Direktiven müssen Sie natürlich auch anwenden. Je nach Assembler ist die Sache mehr oder weniger weit getrieben, ist natürlich auch nicht genormt, sprich, Sie müssen sich mit der Thematik auseinandersetzen und einiges an Lernpensum bewältigen. Haben Sie sich dadurch nicht abschrecken lassen und Ihr Programm schön modularisiert, kommt das nächste Problem.

Vorausgesetzt, Ihr Linker kann beliebig viele Module binden (Vorsicht, einige erlauben nur Eingabezeilen von zum Beispiel 64 Zeichen!), dann ist das natürlich jedesmal eine irre Tipperei, wenn Sie den Linker aufrufen. Dazu sollte es nun eine von zwei Lösungen geben:

Erstens: Der gesamte Lauf wird »im Batch« abgearbeitet (siehe Kapitel 4).

Zweitens: Der Linker bietet eine Anweisung wie »INPUT File-Name«. In diesem Falle schreiben Sie einmal alle Linker-Anweisungen in ein Textfile. Beim Aufruf des Linkers sagen Sie ihm dann, daß er dieses Textfile benutzen soll. Alles in allem, die Sache ist doch recht umständlich. Ich kann Ihnen nur raten, stellen Sie das Thema Code-Module vorerst zurück. Erst wenn Sie den 68000-Assembler richtig beherrschen, und Sie sich an große Aufgaben, wie beispielsweise die Entwicklung eines Basic-Interpreters heranwagen, dann sollten Sie sich – nun allerdings dringend – wieder mit dem Thema Modularisierung auf Code-Ebene auseinandersetzen.

Andererseits ist modulare Programmierung in Assembler der einzige Weg, um auch bei mittelgroßen Programmen einigermaßen über die Runden zu kommen. Wie man dabei praktisch vorgeht, soll das folgende Beispiel zeigen.

Es soll ein Disk-Editor entwickelt werden. Im Hauptmenü hat der Anwender die Auswahl unter den Kommandos »L)esen, E)ditieren, S)chreiben und Exit«. Wie das Menü angeboten wird, lasse ich erst einmal dahingestellt sein. Fest steht nur, daß die Buchstaben L, E, S und X die entsprechende Aktion auslösen sollen. Sozusagen im Bestand habe ich ein Include-File, das eine Taste liest und daraufhin das zugeordnete Unterprogramm aufruft. Dieses File finden Sie in Bild 7.7. Es ist ein Teil des »CASE X OF«-Programms aus Kapitel 5.

```
* start.icl

start   move    #7,-(sp)        ; conin ohne Echo
        trap    #1
        addq.l  #2,sp
        tst     d0              ; kein ASCII-Zeichen?
        beq     start           ; wenn so
        cmpi    #'A',d0         ; kein Buchstabe?
        blt     start           ; dann ignoriere Taste
        bclr    #5,d0           ; Erzwinge Grossbuchstaben
        lea     keys,a0         ; Tabelle gueltige Keys
        move    #count,d1       ; deren Anzahl
```

```
search   cmp.b   (a0)+,d0        ; Key auf aktuellem Platz?
         dbeq    d1,search       ; wenn nicht, weitersuchen bis
                                   Tabellenende
         tst     d1              ; Key gefunden?
         bmi     start           ; wenn nicht, auf ein Neues
         neg     d1              ; sub d1,#count
         add     #count,d1       ; ergibt Platznr. von Key
         lsl     #2,d1           ; die mal 4
         lea     table,a0        ; Adr. der Routine
         move.l  0(a0,d1.w),a0   ; bestimmen
         jsr     (a0)            ; und diese aufrufen
         bra     start           ; u.s.w.
```

Bild 7.7: Startmodul als Include-File

Nachdem diseses Modul existiert, beginne ich nun mein neues Programm so, wie es Bild 7.8 zeigt.

```
         include  "start.icl"
Lese     rts
Edit     rts
Schreib  rts

keys     dc.b     'L','E','S','X'
count    equ      *-keys
         ds.w     0
table    dc.l     Lese, Edit,Schreib,Exit
```

Bild 7.8: Ein neues Programm wird so begonnen

7.5 Top Down, Bottom Up

Sie sehen sofort die Struktur des Programms. Was in den einzelnen Unterprogrammen passiert, interessiert zuerst gar nicht. Man kann nun hergehen und die einzelnen Unterprogramme nacheinander mit »Fleisch füllen«. Ist ein Unterprogramm fertig, wird es ausgetestet. Erst wenn es läuft, wird das nächste begonnen. Praktisch geht man sogar noch einen Schritt weiter. Zum Beispiel benötigt das Unterprogramm »Lese« eine Routine, die einen Sektor liest, und eine weitere, die den gelesenen Sektor auf dem Schirm in hex ausgibt. Dazu brauche ich unter anderem ein Unterprogramm »Anzeige«. Anzeige benötigt aber eine Routine, die ein Wort in die entsprechenden ASCII-Strings umwandelt. Damit ergibt sich dieser Ablauf:

```
Lese     bsr     read_sec
         bsr     anzeige
         rts

read_sec rts

anzeige  bsr     wandle
         bsr     print
         rts

wandle   rts

print    rts
```

Begonnen habe ich ganz oben und bin zum Schluß beim Unterprogramm »print« gelandet. Dies muß ich nun wirklich bearbeiten. Wenn die Routine »print« läuft, kann ich »wandele« beginnen; denn nun erst kann ich ja die gewandelten Hex-Zahlen ausgeben und somit die Routine »print« auch testen. Jetzt werde ich mir »anzeige« vornehmen, das durch mehrfachen Aufruf von Print einen Pufferinhalt auf dem Schirm ausgibt. Danach werde ich »read_sec« schreiben, was diesen Puffer mit Daten füllt. Nun schließlich kann ich im Hauptmenü »Lese« aufrufen und wäre damit wieder ganz oben.

Dieses »von oben nach unten und wieder zurück« nennt man »top down, bottom up«. Dies ist eine Methode der Programmierung, die gerade in Assembler sehr zu empfehlen ist. Sie beschränken damit die Fehlersuche immer nur auf einen kleinen, überschaubaren Bereich. Scheuen Sie dabei auch nicht den Mehraufwand, einzelne Unterprogramme temporär mit Spieldaten zu versorgen. Der Aufwand ist gering, der Nutzen ist groß. Zum Beispiel soll die Routine »wandle« ein Wort in D0 als Hex-String ausgeben. Schreiben Sie dann einfach

```
         move  $19AF,d0
```

vorläufig als erste Zeile im Unterprogramm »wandele«. Wenn Sie nun die Routine testen und »19AF« auf dem Schirm sehen, dann können Sie schon ziemlich sicher sein, daß »wandele« funktioniert.

7.6 Programmentwicklung am Beispiel »bindec«

In diesem Abschnitt soll gezeigt werden, wie man ein Programm Schritt für Schritt entwickelt. Als nützliches Beispiel dient eine Routine, die wir später noch öfter benötigen werden. Ihr Name ist »bindec«. »bindec« soll einen Integerwert, wie ihn der 68000 sieht, also binär, in einen Dezimalstring wandeln, den wir lesen können.

Bild 7.8 zeigt den ersten Schritt, an dem wir die grundsätzliche Technik der Zahlenwandlung studieren wollen

```
* decl
        move    #12345,d2    ; Testzahl

        ext.l   d2           ; Erweitere in Langwort
        divs    #10000,d2    ; 10000er Stelle
        bsr     out          ; ausgeben

        swap    d2           ; Divisionsrest nach d2.w
        ext.l   d2           ; wieder auf Langwort bringen
        divs    #1000,d2     ; nun die 1000er Stelle
        bsr     out

        swap    d2           ; wie vor, die 100er
        ext.l   d2
        divs    #100,d2
        bsr     out

        swap    d2           ; nun die 10er
        ext.l   d2
        divs    #10,d2
        bsr     out

        swap    d2           ; und die 1er
        bsr     out

        move.w  #1,-(sp)     ; mit Funktion CONIN
        trap    #1           ; auf Taste warten
        addq.l  #2,sp        ; Stack korrigieren
        clr     -(sp)        ; Mit TERM
        trap    #1           ; zum Desktop

out     add.b   #'0',d2      ; in ASCII wandeln
        move    d2,-(sp)     ; und gebe aus
        move.w  #2,-(sp)     ; Funktion CONOUT
        trap    #1           ; aufrufen
        addq.l  #4,sp        ; Stack korrigieren
        rts
```

Bild 7.9: Das Prinzip von »bindec«

Das Prinzip der Zahlenwandlung ist ganz einfach. Wir müssen die Zahl, beispielsweise 123, ausdrücken als 1 Hunderter, 2 Zehner und 3 Einer. Wenn wir so die Ziffern 1, 2 und 3 isoliert haben, sind das »im Computer« zwar immer noch Zahlen, aber dann muß man darauf nur noch den ASCII-Code des Zeichens '0' addieren, und schon hat man druckbare Zeichen. Die Methode der Isolation der einzelnen Ziffern ist die fortlaufende Division und zwar so

123	/ 100	= 1	Rest	23
23	/ 10	= 2	Rest	3
3	/ 1	= 3	Rest	0

Das Dividieren wird beim 68000 mit Hilfe der Befehle

```
DIVS oder DIVU
```

erledigt. Das heißt »Division Signed« (mit Vorzeichen) oder »Division Unsigned« (ohne Vorzeichen). Dividiert wird ein 32- Bit-Dividend durch einen 16-Bit-Divisor. Dividiert wird immer Ziel/Quelle. Danach steht das Ergebnis im Zieloperanden und zwar so:

höherwertiges Wort	niederwertiges Wort
Rest	Quotient

Das Programm von Bild 7.9 soll das Wort im Register D2 in »Dezi« wandeln. Da vom DIVS-Befehl ein Langwort-Dividend erwartet wird, muß das Wort in D2 mittels des »ext.l«-Befehls auf Langwort »extended« (erweitert) werden. Nun wird D2 durch 10000 dividiert. Das Ergebnis ist der Wert der 10000er Stelle, der im Unterprogramm »out« ausgegeben wird. Nun wird mittels des SWAP-Befehls der Rest in das niederwertige Wort gebracht, dieser wird wieder »auf Langwort« erweitert und dann durch 1000 dividiert. Das setzt sich dann so mit der 100er und der 10er Stelle fort. Beim »Einer« müssen wir natürlich nicht mehr dividieren, nur vergessen dürfen wir ihn nicht. So weit so gut, nur wenn man sich das Programm so ansieht, fallen doch einige Wiederholungen auf. Da muß man doch rationalisieren können! Den ersten Ansatz dazu zeigt Bild 7.10.

```
* dec2
        move    #12345,d2     ; Testzahl

        move    #10000,d1     ; 10000er wandeln
        bsr     out2          ; und ausgeben
        move    #1000,d1      ; nun die 1000er Stelle
        bsr     out1
        move    #100,d1       ; die 100er
        bsr     out1
        move    #10,d1        ; die 10er
        bsr     out1

        move    #1,d1         ; die 1er
        bsr     out1

        move.w  #1,-(sp)      ; mit Funktion CONIN
        trap    #1            ; auf Taste warten
        addq.l  #2,sp         ; Stack korrigieren
        clr     -(sp)         ; Mit TERM
        trap    #1            ; zum Desktop

out1    swap    d2            ; Divisionsrest nach d2.w
```

```
out2    ext.l   d2          ; wieder auf Langwort bringen
        divs    d1,d2
        add.b   #'0',d2     ; wandle in ASCII
        move    d2,-(sp)    ; und gebe aus
        move.w  #2,-(sp)    ; Funktion CONOUT
        trap    #1          ; aufrufen
        addq.l  #4,sp       ; Stack korrigieren
        rts
```

Bild 7.10: Ratio-Schritt 1: Mehr Arbeit ins Unterprogramm

Sie sehen, die Befehle »swap«, »ext.l« und »divs« sind in das Unterprogramm gewandert. Da beim ersten Aufruf aber nicht »geswappt« werden darf, wurde das Unterprogramm mit zwei Einsprungstellen versehen. Das ist ein beliebter, aber sehr unfeiner Trick. Der Divisor wird jeweils im Register D1 übergeben. Nun stört noch die Tatsache, daß da 5 nahezu gleiche Unterprogramm-Aufrufe existieren. Wie man das ändert, zeigt Bild 7.11

```
* dec3
        move    #12345,d2   ; Testzahl

        move    #10000,d1   ; erster Divisor
        bsr     out2        ; und ausgeben
        move    #3,d3
loop    divs    #10,d1
        bsr     out1
        dbra    d3,loop

        move.w  #1,-(sp)    ; mit Funktion CONIN
        trap    #1          ; auf Taste warten
        addq.l  #2,sp       ; Stack korrigieren
        clr     -(sp)       ; Mit TERM
        trap    #1          ; zum Desktop

out1    swap    d2          ; Divisionsrest nach d2.w
out2    ext.l   d2          ; wieder auf Langwort bringen
        divs    d1,d2
        add.b   #'0',d2     ; wandle in ASCII
        move    d2,-(sp)    ; und gebe aus
        move.w  #2,-(sp)    ; Funktion CONOUT
        trap    #1          ; aufrufen
        addq.l  #4,sp       ; Stack korrigieren
        rts
```

Bild 7.11: Ratioschritt 2: Eine Schleife hinzu

Die Schleife wurde nach altbekannter Art mittels »dbra« aufgebaut, wobei D3 als Zähler dient. Innerhalb der Schleife wird nun der Divisor D1 selbst immer durch 10 dividiert. Da die Schleife bis zum »Einer« laufen muß, wird dieser zum Schluß überflüssigerweise durch Eins dividiert. Das abzufangen kostet aber mehr Zeit, also lassen wir das so stehen. Nun meine ich aber, daß das Unterprogramm überflüssig ist. Wie man das UP in die Schleife bringt, zeigt Bild 7.12

```
* dec4
        move    #123,d2       ; Testzahl

        move    #10000,d1     ; erster Divisor
        move    #4,d3         ; Schleifenzaehler
        bra     out2          ; 10000er wandeln
loop    divs    #10,d1        ; und nun die 1000er bis 1er
out1    swap    d2            ; Divisionsrest nach d2.w
out2    ext.l   d2            ; wieder auf Langwort bringen
        divs    d1,d2         ; naechste Stelle
        add.b   #'0',d2       ; wandle in ASCII
        move    d2,-(sp)      ; und gebe aus
        move.w  #2,-(sp)      ; Funktion CONOUT
        trap    #1            ; aufrufen
        addq.l  #4,sp         ; Stack korrigieren
        dbra    d3,loop
*       auf Taste warten und TERM, wie gehabt
```

Bild 7.12: Ratioschritt 3: Unterprogramm entfällt

Beachten Sie, daß ich wegen des beim ersten Mal unerwünschten »swap« jetzt in die Schleife hineinspringe, weshalb ich nun den Schleifenzähler mit 4 initialisieren muß. Jetzt aber genug der Ratio, kümmern wir uns lieber um die Schönheit. Ihnen ist sicherlich aufgefallen, daß das Programm führende Nullen schreibt, wenn die Zahlen kleiner als fünfstellig sind. Die unterdrückt man üblicherweise durch Ausgabe von Blanks anstatt führender Nullen. Wie man das macht, zeigt Bild 7.13.

```
* dec5
        move    #1001,d2      ; Testzahl

        move    #10000,d1     ; erster Divisor
        move    #4,d3         ; Schleifenzaehler
        clr     d4            ; Flag Nullunterdrueckung
        bra     out2          ; 10000er wandeln
```

```
loop    divs    #10,d1      ; und nun die 1000er bis 1er
out1    swap    d2          ; Divisionsrest nach d2.w
out2    ext.l   d2          ; wieder auf Langwort bringen
        divs    d1,d2       ; naechste Stelle
        add.b   #'0',d2     ; wandle in ASCII

        cmpi.b  #'0',d2     ; ist es eine Null? +++ neu +++
        bne     out3        ; wenn nicht ausgeben
        tst     d4          ; Blank erlaubt?
        bne     out3        ; nein, gebe die Null aus
        move    #' ',d2     ; setze Blank ein
        bra     out4        ; und gebe aus
out3    move    #1,d4       ; Flag keine Blanks mehr

out4    move    d2,-(sp)
        move.w  #2,-(sp)    ; Funktion CONOUT
        trap    #1          ; aufrufen
        addq.l  #4,sp       ; Stack korrigieren
        dbra    d3,loop

*       auf Taste warten und TERM, wie gehabt
```

Bild 7.13: Schritt 4: Unterdrückung führender Nullen hinzu

Das Problem ist einfach zu beschreiben. Führende Nullen sollen durch Blanks ersetzt werden, andere Nullen natürlich nicht. Dazu benötigt man ein Flag (Merker), das gesetzt wird, sobald eine Zahl ungleich Null auftaucht. Soweit die Logik. Praktischer ist jedoch, den Gedankengang etwas abzuwandeln, nämlich so: Jede Zahl ungleich Null setzt das Flag. Damit kann man sich die sehr aufwendige Realisierung des Unterscheidens von erster »Nicht-Null« und anderen Nullen ersparen.

In unserem Fall ist das Register D4 das Flag. Die Abfrage beginnt in der Zeile mit »+++ neu +++« am rechten Rand. Ist die Zahl keine Null, wird D4 mit 1 geladen und dann das Zeichen ausgegeben. Ansonsten muß es eine Null sein, und nun kommt der Test. Ist D4 gesetzt, wird die Null als Null ausgedruckt. Wenn nicht, wird D2 mit einem Blank geladen.

Nun habe ich beschlossen, das Ganze soll ein universell verwendbares Unterprogramm werden. Das heißt zuerst, das Unterprogramm darf die Zeichen nicht auf dem Schirm ausgeben, weil man sonst zum Beispiel nicht drucken kann. Die Änderung ist kein Problem. Bild 7.14 bringt die Lösung. Die Ausgabe erfolgt in einen Puffer namens »buffer«. Als Zeiger durch den Puffer wirkt das Register A0. Um zu wissen, wieviel Zeichen im Puffer gültig sind, wird als letztes Zeichen ein Null-Byte geschrieben. Das ist sehr praktisch, haben wir doch damit gleichzeitig einen TOS-String.

```
* dec6
        move    #1001,d2     ; Testzahl

        move    #10000,d1    ; erster Divisor
        move    #4,d3        ; Schleifenzaehler
        clr     d4           ; Flag Nullunterdrueckung
        lea     buffer,a0    ; Ergebnispuffer
        bra     out2         ; 10000er wandeln
loop    divs    #10,d1       ; und nun die 1000er bis 1er
out1    swap    d2           ; Divisionsrest nach d2.w
out2    ext.l   d2           ; wieder auf Langwort bringen
        divs    d1,d2        ; naechste Stelle
        add.b   #'0',d2      ; wandle in ASCII

        cmpi.b  #'0',d2      ; ist es eine Null?
        bne     out3         ; wenn nicht ausgeben
        tst     d4           ; Blank erlaubt?
        bne     out3         ; nein, gebe die Null aus
        move    #' ',d2      ; setze Blank ein
        bra     out4         ; und gebe aus
out3    move    #1,d4        ; Flag keine Blanks mehr

out4    move    .b d2,(a0)+ ; Zeichen -> Puffer
        dbra    d3,loop
        move.b  #0,(a0)      ; Abschlusszeichen

        pea     buffer       ; String ausgeben
        move    #9,-(sp)     ; mittels PTTLIN-Funktion
        trap    #1
        addq.l  #6,sp

*       auf Taste warten und TERM, wie gehabt

        bss
buffer  ds.b    6            ; Puffer fuer Ergebnis
```

Bild 7.14: Schritt 5: Ausgabe in einen String

Nur, ein richig universelles Unterprogramm ist es damit trotzdem noch nicht. Störend ist schon, daß die Wertübergabe im Register D2 erfolgt, das könnte ja das Hauptprogramm benutzen. Schlimm jedoch ist, daß eine Variable mit dem Namen »buffer« fest mit diesem Programm gekoppelt ist. Das muß geändert werden. Wie, zeigt Bild 7.15.

```
* dec 7
        move    #1001,-(sp) ; Testzahl
        pea     buffer      ; Ergebnis-Puffer
        bsr     bindec      ; Routine aufrufen

        pea     buffer      ; String ausgeben
        move    #9,-(sp)    ; mittels PRTLIN-Funktion
        trap    #1
        addq.l  #6,sp

*       auf Taste warten und TERM, wie gehabt
;-----------------------------------------------------------
bindec  move.l  4(sp),a0    ; Pufferadresse holen
        move    8(sp),d2    ; und die zu wandelnde Zahl
        move    #10000,d1   ; erster Divisor
        move    #4,d3       ; Schleifenzaehler
        clr     d4          ; Flag Nullunterdrueckung
        lea     buffer,a0   ; Ergebnispuffer
        bra     out2        ; 10000er wandeln
loop    divs    #10,d1      ; und nun die 1000er bis 1er
out1    swap    d2          ; Divisionsrest nach d2.w
out2    ext.l   d2          ; wieder auf Langwort bringen
        divs    d1,d2       ; naechste Stelle
        add.b   #'0',d2     ; wandle in ASCII

        cmpi.b  #'0',d2     ; ist es eine Null?
        bne     out3        ; wenn nicht ausgeben
        tst     d4          ; Blank erlaubt?
        bne     out3        ; nein, gebe die Null aus
        move    #' ',d2     ; setze Blank ein
        bra     out4        ; und gebe aus
out3    move    #1,d4       ; Flag keine Blanks mehr

out4    move.b  d2,(a0)+    ; Zeichen -> Puffer
        dbra    d3,loop
        move.b  #0,(a0)     ; Abschlusszeichen

        move.l  (sp)+,a0    ; hole Return-Adresse
        addq.l  #6,sp       ; Parameter vom Stack
        jmp     (a0)        ; und return

        bss
buffer  ds.b    6           ; Puffer fuer Ergebnis
```

Bild 7.15: Schritt 6: Parameterübergabe über den Stack

Aufgerufen wird das Unterprogramm »decbin« in der Folge

Wert auf den Stack
Pufferadresse auf den Stack
bsr decbin

Danach befinden sich auf dem Stack bei

8(sp) : Wert
4(sp) : Pufferadresse
0(sp) : Return-Adresse

Folglich kann man sich mit

```
move.l  4(sp),a0
move.w  8(sp),d2
```

diese Parameter leicht holen. Der Rest läuft dann wie bekannt, allerdings mit einem kleinen Unterschied zum Schluß.

```
move.l  (sp)+,a0
```

holt die Return-Adresse vom Stack und erhöht den Stackpointer um 4. Nun muß ich aber noch die 6 Bytes der Parameter (Wort für Wert und Langwort für Adresse) vom Stack »entfernen«. Das geschieht mittels der Anweisung

```
addq.l  #6,sp
```

Zum Schluß muß man natürlich »returnen«, was aber jetzt nur noch heißt, »springe zur Return-Adresse«, also

```
jmp     (a0)
```

Der Mechanismus kommt Ihnen bekannt vor? Sie haben recht, so ähnlich arbeitet das TOS, wenn wir es mittels TRAP #n aufrufen. Wenn Sie das »addq.l #6,sp« weglassen, müssen Sie dies nach dem Aufruf erledigen. Da hätten Sie dann das wahre TOS-Feeling.

So weit so gut, aber perfekt sind wir immer noch nicht. Unser »bindec« zerstört leider die Register D1 bis D4. A0 wird zwar auch geändert, aber das ist üblich, auch D0 ist immer »scratch« (Schmierpapier). Im letzten Schritt, dem im Bild 7.16, soll das auch abgestellt werden.

```
bindec  movem.l  d1-d4,-(sp)  ; benutzte Register retten
        move.l   20(sp),a0    ; Pufferadresse holen
        move     24(sp),d2    ; und die zu wandelnde Zahl

        move     #10000,d1    ; erster Divisor
        move     #4,d3        ; Schleifenzaehler
        clr      d4           ; Flag Nullunterdrueckung
        lea      buffer,a0    ; Ergebnispuffer
        bra      bd3          ; 10000er wandeln
```

```
bd1     divs     #10,d1        ; und nun die 1000er bis 1er
bd2     swap     d2            ; Divisionsrest nach d2.w
bd3     ext.l    d2            ; wieder auf Langwort bringen
        divs     d1,d2         ; naechste Stelle
        add.b    #'0',d2       ; wandle in ASCII

        cmpi.b   #'0',d2       ; ist es eine Null?
        bne      bd4           ; wenn nicht ausgeben
        tst      d4            ; Blank erlaubt?
        bne      bd4           ; nein, gebe die Null aus
        move     #' ',d2       ; setze Blank ein
        bra      bd5           ; und gebe aus
bd4     move     #1,d4         ; Flag keine Blanks mehr

bd5     move.b   d2,(a0)+      ; Zeichen -> Puffer
        dbra     d3,bd1
        move.b   #0,(a0)       ; Abschlusszeichen
        movem.l  (sp)+,d1-d4   ; Register zurueck
        move.l   (sp)+,a0      ; hole Return-Adresse
        addq.l   #6,sp         ; Parameter vom Stack
        jmp      (a0)          ; und return
```

Bild 7.16: Der letzte Schritt: Arbeits-Register werden gesichert

Vorab, ich habe die Labels so geändert, daß Sie hoffentlich in anderen Programmen nicht vorkommen. Falls Ihr Assembler lokale Labels bietet, dann sollten Sie davon Gebrauch machen (bei Metacomco $n...). Neu ist der Befehl

```
movem.l   d1-d4,-(sp)
```

Damit kann eine ganze Gruppe oder Liste von Registern auf den Stack gebracht werden. Auch Schreibweisen wie

```
movem.l   d1-d4/a1-a2/a5,-(sp)
```

sind zugelassen. Das Gegenstück (vom Stack holen) lautet dann

```
movem.l   (sp)+,d1-d4/a1-a2/a5
```

Da sich in unserem Beispiel nun 4 Register, also 16 Bytes auf dem Stack zusätzlich befinden, müssen wir, um die Parameter zu holen, auch diese 16 Bytes zugeben, daher:

```
move.l   20(sp),a0     ; Pufferadresse holen
move     24(sp),d2     ; und die zu wandelnde Zahl
```

Nun speichern Sie bitte die Routine »bindec« in einem extra File, wir werden sie später noch benötigen.

Kapitel 8

Das File-System des Atari ST

Directory

Directory-Entries

DTA-Puffer

Boot-Sektor

BIOS Parameter Block

FAT und Cluster

Files einrichten,

lesen,

schreiben

und kopieren

Programm-Files

8.1 Überblick

Unter File-System versteht man alles, was mit den Dateien (Files) auf Disketten und Festplatten zu tun hat.

Da man praktisch in keiner Programmiersprache daran vorbeikommt, Daten und Programme auf Disketten zu schreiben bzw. von ihnen zu lesen, müssen wir uns auch in Assembler mit dieser Thematik befassen. Naturgemäß auch hier etwas tiefgehender als in den Hochsprachen, wo das Thema mit ein paar Disk-Befehlen erledigt ist.

Eine normale ST-Diskette ist in 80 Spuren eingeteilt. Diese Spuren nennt man Tracks. Jeder Track ist ein Ring auf der Disk. Da die Einheit Track etwas groß ist, teilt man jeden Track in 9 Sektoren. Jeder Sektor faßt 512 Bytes. Das ergibt 80 x 9 x 512 = 368 640 Bytes pro Diskette. Bei zweiseitigen Disketten verdoppelt sich das Ganze. Die Seiten sind mit 0 und 1 numeriert. Die Tracks werden von 0 bis 79 gezählt, die Sektoren in Abweichung von dieser Regel hingegen von 1 bis 9. Wenn Sie einen Sektor direkt ansprechen wollen – dafür gibt es XBIOS-Funktionen – müssen Sie in der eben genanten Zählweise Seite (Side), Track und Sektor angeben.

GEMDOS selbst zählt intern etwas anders. Da sind die Sektoren von 0 bis 719 (einseitige Diskette) bzw. von 0 bis 1439 (doppelseitig) durchnumeriert. Dies sind die sogenannten relativen Sektoren. Die ersten 18 Sektoren (relativ 17) sind auf jeder Diskette für »Verwaltungs-Aufgaben« belegt. Hier vorab ein Überblick, Details kommen später.

Sektor(en)	Inhalt
0	Boot-Sektor
1-5	FAT 1
6-10	FAT 2
11-17	Directory
18-Ende	Daten

Das ist jedenfalls die Standardeinteilung. Es steht Ihnen frei – in gewissen Grenzen – daran »zu drehen«.

Wie die Diskette tatsächlich konfiguriert ist, steht im Boot-Sektor. Deshalb muß dieser Bootsektor auf jeder Diskette vorhanden sein, auch wenn die Disk gar kein Boot-Programm enthält.

Der Datenbereich beginnt also mit Sektor 18, und ab da zählt GEMDOS noch auf eine zweite Art und Weise, nämlich mittels der schönen Einheit Cluster. Ein Cluster ist eine Menge von normalerweise 2 Sektoren (auch das kann geändert werden) oder 1024 Bytes. Ein Cluster ist die kleinste Einheit, die einem File zugewiesen wird. D.h., wenn Sie eine Datei mit 1025 Bytes belegen, wird sie 2 Cluster oder 2048 Bytes auf der Diskette verbrauchen. Bei zweiseitigen Disketten werden die Tracks sozusagen abwechselnd auf Seite 0 und 1 beschrieben. Das hat zur Folge, daß Sie dort beim Umrechnen von relativen Sektoren auf absolute Track/Sektor-Nummern folgendes Schema beachten müssen:

Rel. Sektor	Seite	Track
0 - 8	0	0
9 - 17	1	0
18 - 26	0	1

Der Vorteil dieser Methode ist deutlich. Da für jeweils 18 Sektoren der Schreib/Lesekopf nicht neu positioniert werden muß – ein aus Sicht des Computers sehr langsamer Vorgang – sind zweiseitige Disketten schneller als einseitige.

8.2 Das Directory

Der normale Zugriff auf eine Datei erfolgt bekanntlich über File-Namen. Damit das System diese Namen in Track- und Sektornummern umsetzen kann, gibt es das sogenannte Directory, was man in erster Näherung als Inhaltsverzeichnis übersetzen könnte. Tatsächlich ist es vom Directory bis zur Sektornummer noch ein weiter Weg, aber hier ist der Einstieg. Aus der Einleitung wissen Sie noch, daß für das Directory die relativen Sektoren 11-17 reserviert sind. Jeder dieser 7 Sektoren ist in Blöcke von 32 Bytes Länge eingeteilt, das ergibt 16 Blöcke je Sektor bzw. 16 x 7 = 112 Blöcke je Directory.

8.3 Directory-Entries

Die eben geschilderten Blöcke nennt man Directory-Entries. So ein Entry ist praktisch eine Schublade mit Platz für genau einen Filenamen nebst »Zubehör«. Die 32 Bytes eines Entry werden von 0 bis 31 gezählt, hier ein Überblick:

Byte	Inhalt
0	Flag oder erster Buchstabe des Namens
0-7	File-Name
8-10	Extension (zum Beispiel PRG oder TOS)
11	Attribut
12-21	reserviert
22-23	Uhrzeit
24-25	Datum
26-27	Start-Cluster
28-31	File-Göße in Bytes

Nun der Reihe nach:

Byte 0: Das Flag-Byte ist sozusagen ein Blitzinfo für das TOS.
Hier bedeuten:
$00: Entry ist frei, war noch nie benutzt.
$E5: Entry ist frei, weil gelöscht
§2E: Entry verweist auf ein Sub-Directory. Ein Sub-Directory ist ein Ordner auf dem Schreibtisch. Sub-Directories werden im Datenbereich angelegt!

Alle anderen Zeichen: Erster Buchstabe des Namens.

Byte 0-7: File-Name

Byte 8-10: Namenszusatz

Byte 11: Attribut. Hier gilt

Wert	Bedeutung
$00	Lesen und Schreiben erlaubt
$01	Nur Lesen erlaubt
$02	Versteckte Datei
$04	System-Datei (automatisch versteckt)
$08	Es gibt einen Disketten-Namen (in Bytes 0-10)
$10	Der Eintrag zeigt auf ein Sub-Directory
$20	Archiv-Bit (zur Zeit nicht benutzt)

Da jedes der Bit 0-5 signifikant ist, dürfen die Werte auch addiert werden.

Byte 22-23: Uhrzeit. In der Reihenfolge 23/22 ist dies ein Wort mit der Bedeutung:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	h	h	h	h	h	m	m	m	m	m	m	s	s	s	s	s

Da sich mit 5 Bit nur die Zahlen 0-31 darstellen lassen, sind diese die Sekunden mal zwei.

Byte 24-25: Datum. In der Reihenfolge 25/24 ist dies ein Wort mit der Bedeutung:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	j	j	j	j	j	j	j	m	m	m	m	t	t	t	t	t

Zum Jahr muß immer 1980 addiert werden.

Byte 26-27: Start-Cluster. Muß als Wort in der Folge 27/26 gelesen werden.

Byte 28-31: File-Größe in Bytes.

Hier wird es langsam Zeit, die Ungereimtheiten zu erklären. Die Disketten-Organisation ist sehr stark an das MS-DOS des IBM-PC angelehnt. Das geht soweit, daß die bei der 8088-CPU übliche Vertauschung des niederwertigen Bytes (low byte) und höherwertigen Bytes

(high byte) auch hier stattfindet. Entsprechendes gilt für Worte in Langworten. Demnach sind die Bytes 28-31 in der Folge

low word(low,high), higword (low,high)

zu lesen. Das Einfachste ist wohl aus 68000-Sicht, man liest die Bytes arabisch (von rechts nach links).

8.4 Der DTA-Puffer

Man kann nun das Directory mit seinen sieben Sektoren in den RAM einlesen und sich dann das Ganze als eine Tabelle denken, durch die man sich mit »Step 32« hindurcharbeiten kann. Es geht aber einfacher. Das TOS legt bei bestimmten Operationen (die wir in Kürze anwenden) die wesentlichen Informationen zu einem File in einem Puffer ab, der DTA-Buffer (Disk Transfer Address) heißt. Die DTA hat folgende Gliederung:

Byte	Bedeutung	
0-20	reserviert	
21	Attribut	(wie Byte 11 im Directory-Entry)
22-23	Uhrzeit	
24-25	Datum	(wie im Directory-Entry)
26-29	File-Größe	(wie Byte 28-31im Directory-Entry)
30-43	Name und Extension des Files	

8.5 Directory ausgeben

Mit Hilfe zweier GEMDOS-Funktionen wollen wir nun das Directory einer Diskette lesen. Bild 8.1 zeigt das Listing.

```
* File dir. s

GEMDOS   equ     1          ; die Nummer fuer den GEMDOS-
                              Trap
CONIN    equ     1          ; Funktion auf Taste warten
PRTLIN   equ     9          ; Funktion String ausgeben
SETDTA   equ     $1A        ; Setze Disk Transfer Adresse
SFIRST   equ     $4E        ; Search First (suche erstes)
                              File
SNEXT    equ     $4F        ; Suche naechstes File
```

```
        text                    ; Programm-Segment

        move.l  #msg,d0         ; Adresse des Strings ueberge-
                                  ben
        bsr     print           ; String drucken

        pea     dta             ; Adresse des DTA-Puffers
        move.w  #SETDTA,-(sp)   ; Funktionsnummer
        trap    #GEMDOS
        addq.l  #6,sp           ; Stack korrigieren

        move.w  #0,-(sp)        ; Attribut "normale Files"
        pea     fname           ; Adresse des File-Namens
        move.w  #SFIRST,-(sp)   ; Funktionsnummer
        trap    #GEMDOS
        addq.l  #8,sp
loop    tst     d0              ; File gefunden?
        bne     done            ; wenn nicht, fertig
        move.l  #name,d0        ; sonst...
        bsr     print           ; ... Name drucken

        move.w  #SNEXT,-(sp)    ; naechstes File suchen
        trap    #GEMDOS
        addq.l  #2,sp
        bra     loop

done    move.l  #wait,d0        ; Text "Warte auf Taste"
        bsr     print
        move    #CONIN,-(sp)
        trap    #GEMDOS         ; und warte
        addq.l  #2,sp
        clr     -(sp)
        trap    #GEMDOS         ; zum Desktop

print   move.l  d0,-(sp)        ; String ausgeben
        move.w  #PRTLIN,-(sp)
        trap    #GEMDOS
        addq.l  #6,sp

        move.l  #crlf-(sp)      ; und noch ein CR/LF
        move.w  #PRTLIN,-(sp)
        trap    #GEMDOS
        addq.l  #6,sp
        rts

        data
msg     dc.b    27,'E'
        dc.b    'Directory'
crlf    dc.b    13,10
        dc.b    0
```

```
        ds.w    0
wait    dc.b    13,10,'Warte auf Taste',0

        ds.w    0
fname   dc.b    '*.*',0

        ds.w    0
dta     ds.b    30
name    ds.b    14

        end
```

Bild 8.1: Lesen des Directory

Das Programm von Bild 8.1 bringt einiges Neue. Zuerst werden mittels der EQU-Direktiven einige Konstanten definiert. Das ist zwar in gewissen Grenzen überflüssig, denn natürlich wissen Sie und ich, daß der Trap #1 der GEMDOS-Aufruf ist und sich »trap #1« viel schneller schreibt als »trap #GEMDOS«, aber: die Funktion SFIRST ist schon nicht mehr so bekannt. Da ist es ganz praktisch, wenn ich sie einmal zu Anfang erkläre und dann im Programm nie wieder. Der ursprüngliche Sinn dieser Konstanten-Definition ist allerdings ein anderer. Wenn nämlich eine häufig im Programm benutzte Konstante geändert werden soll, ist es vorteilhaft, wenn man nur die eine EQU-Zeile anpassen muß. Dieser Fall trifft hier nicht zu.

Die Funktion SFIRST heißt »Search First«. Gesucht werden soll das erste Auftreten eines File-Namens. Der Trick ist nun, daß im File-Namen auch die Ersatzzeichen »*« und »?« erlaubt sind. Der Stern steht für »ganzer Name« oder »ganze Extension«. Um also alle Files zu suchen, muß man nur als Name »*.*« angeben. Außerdem ist noch das Fragezeichen anstelle eines beliebigen Buchstabens erlaubt. Der Funktion SFIRST muß die Adresse eines Strings mit dem Namen übergeben werden. Hat sie den Namen oder ein erstes Auftreten entsprechend der Maske (*.*) gefunden, schreibt sie ihn in den DTA-Puffer und setzt das Register D0 auf Null. Im Falle »nicht gefunden« wird D0 = -1. Danach kommt die Funktion SNEXT zum Tragen. »Search Next« (Suche nächsten) entspricht einem erneuten Aufruf von SFIRST, nur daß hierzu nicht mehr die Adresse des Suchsstrings übergeben werden muß. Nun müssen wir nur noch wissen, wo der DTA-Puffer steht. Dafür benutzen wir die FUNKTION SETDTA. Dieser Funktion übergibt man die Adresse eines Puffers, und GEMDOS wird dann ab sofort diesen Puffer als DTA benutzen. Am Ende des Listings sehen Sie diesen Puffer. Ich hätte auch einfach schreiben können »dta ds.b 44«.

Ich habe aber durch die Aufteilung in »dta ds.b 30« und »dta ds.b 44« die Marke »name« erhalten. Nun kann ich den Filenamen mit »name« ansprechen, was vielleicht etwas klarer ist als »dta+30«. Ansonsten läuft das Programm in einer Schleife der Art

```
        suche erstes Mal
loop    gefunden?
        wenn nicht, springe nach done
        Namen ausgeben
```

```
                suche nächsten Namen
                gefunden?
                wenn nicht, springe nach done
                bra loop
        done    hier sind wir fertig
```

Das Listing sieht ja nicht schlecht aus, aber wie gefällt Ihnen die Lösung von Bild 8.2?

```
        writes  msg         ; Text "Directory"
        writes  crlf        ; noch ein CR/LF
        setdta  dta         ; DTA einrichten
        sfirst  fname       ; Erstes File gefunden?
        bne     done        ; wenn nicht, fertig
loop    writes  name        ; sonst ...
        writes  crlf        ; ... Name drucken
        snext               ; Neuer Versuch
        beq     loop        ; wenn noch einer
done    writes  wait        ; Text "Warte auf Taste "
        conin
        term
        end
```

Bild 8.2: Directory-Listing mittels Makros

Der nicht mehr aufgeführte Datenbereich entspricht dem von Bild 8.1. Ansonsten sehen Sie hier die recht konsequente Anwendung von Makros. Sie können sich nun einiges an Tipparbeit ersparen, wenn Sie in das Listing von Bild 8.2 als erste Zeile einfügen

```
        include "gemdos.mac"
```

Nun beginnen Sie mit der Anlage Ihrer Makro-Bibliothek, indem Sie die Makros aus Bild 8.3 als Text eingeben und unter »gemdos.mac« abspeichern. Ist das geschehen, können Sie das Programm von Bild 8.2 auch assemblieren. Nur noch ein Hinweis: Da »Mac-Libs« ja doch ganz schön lang werden kann, hat man davon eigentlich sogar zwei Exemplare. Im ersten ist jeder Makro schön auskommentiert. Dieses Listing liegt ausgedruckt sozusagen als Handbuch neben dem Computer. Im zweiten Listing fehlen sämtliche Kommentare. Mit diesem File wird praktisch gearbeitet.

```
* Auszug aus GEMDOS.MAC
*
* Achtung: je nach Assembler \n durch ?n oder xn
* ersetzen!

term    macro
        clr     -(sp)       ; Funktion Term(inate)
        trap    #1
        endm

conin   macro
        move    #1,-(sp)    ; Funktion CONIN
        trap    #1
        addq.l  #2,sp
        endm

writes  macro
        pea     \1
        move.w  #9,-(sp)    ; Funk. PRINTLINE
        trap    #1
        addq.l  #6,sp
        endm

setdta  macro
        pea     \1          ; Adresse des DTA-Puffers
        move.w  #$1A,-(sp)  ; Funktionsnummer
        trap    #1          ; Funktion aufrufen
        addq.l  #6,sp       ; Stack korrigieren
        endm

sfirst  macro
        move.w  #0,-(sp)    ; Attribut "normale Files"
        pea     \1          ; Adresse des File-Namens
        move.w  #$4E,-(sp)  ; Funktionsnummer
        trap    #1          ; Aufruf
        addq.l  #8,sp       ; Stackkorrektur
        tst     d0          ; hole Status
        endm

snext   macro
        move.w  #$4F,-(sp)  ; naechstes File suchen
        trap    #1
        addq.l  #2,sp
        tst     d0
        endm
```

Bild 8.3: Die ersten Anfänge von »GEMDOS.MAC«

8.6 Boot-Sektor und BIOS-Parameter-Block

Zur Auflockerung will ich einmal erzählen, warum Boot Boot heißt. Wie sich bei uns Münchhausen an den eigenen Haaren hochzieht, so macht dieses das amerikanische Pendant an den Bootstraps (Schnürsenkeln). Wenn ein Computer startet, zieht er sich auch selbst hoch. Zuerst kommt ein Stück Programm im ROM zur Wirkung, das einen Sektor auf der Diskette liest. In diesem Sektor steht ein Programm, in dem steht, wie man ein File liest. Dieses Programm lädt dann das Betriebssystem von der Diskette. Für den ST gilt dies, solange das TOS nicht im ROM ist. Der erste Sektor auf der Diskette (Track 0, Sektor 1), bei dem dieses »Booten« startet, ist der Boot-Sektor. Falls es eine bootfähige Diskette ist, steht da als erste Anweisung »springe zum Boot-Code«, ansonsten stehen aber im Boot-Sektor noch eine Menge anderer Informationen, die das TOS braucht, um eine Diskette zu identifizieren. Diese Daten nun der Reihe nach:

Byte	Bedeutung
$00	Ggf. »BRA Boot-Code«
$02	Zeichenfolge 'Loader', wenn bootfähig
$08	Seriennummer (Zufallszahl vom Formatieren)
$0B	Sektor-Größe in Bytes (Beginn BPB-Daten)
$0C	Sektoren pro Cluster
$0D	Clustergröße in Bytes
$0E	Reservierte Sektoren
$10	Anzahl FAT's
$11	Anzahl Directory-Entries
$13	Sektoren auf der Disk
$15	unbenutzt
$16	Sektoren pro FAT
$18	Sektoren pro Track
$1A	Anzahl Zylinder (Seiten) der Disk
$1C	Anzahl versteckte Sektoren (Ende BPB)
	Beginn Boot-Daten
$1E	Flag, wenn Command.Prg anstatt Desktop
$20	Flag File- oder Sektor-Boot
$22	Start-Sektor und
$24	Anzahl der zu ladenden Sektoren
$26	Adresse, ab der geladen werden soll
$2A	Adresse des FAT-Puffers
$2E	Name, wenn File-Boot (TOS.IMG)
$03A	Beginn Boot-Programm
$1FE	Checksumme

Die Daten von $0B bis $1C sind für jede Diskette lebenswichtig, andernfalls kann das TOS sie nicht identifizieren. Deshalb baut sich das TOS aus diesen Daten den sog. BIOS-Parameter-Block (BPB) auf, sobald die Diskette geöffnet wird. Auf der Diskette werden alle Daten im IBM-PC-Format gespeichert, was auch heißt, daß bei den Zahlen ab Wortgröße wieder das höherwertige und das niederwertige Byte vertauscht sind. Der im RAM angelegte BPB ist allerdings schon im 68000-Format notiert. Da das TOS ständig den BPB benötigt, gibt es auch eine BIOS-Funktion, die einen Zeiger auf den BPB zurückgibt.

8.7 Ausgabe des BPB

Der BPB im RAM selbst ist eine Folge von 9 Worten mit folgendem Inhalt:

Wort	Bedeutung	Wert (einseitige Diskette)
0	Sektorgröße in Bytes	512
1	Sektoren pro Cluster	2
2	Cluster-Größe in Bytes	1024
3	Sektoren im Directory	7
4	Sektoren pro FAT	5
5	Start-Sektor von FAT 2	6
6	Start-Sektor Daten	18
7	Anzahl Daten-Cluster	351
8	Diskette / Harddisk	0 / 1

Mit Bild 8.4 folgt ein Programm, das genau diese BPB-Tabelle ausgibt. Nur auf die letzte Zeile habe ich verzichtet.

Das Programm »lebt« von der BIOS-Funktion #7. Diese gibt in D0 einen Zeiger auf den BPB zurück.

```
* BPB    BIOS-Parameter-Block ausgeben

        move      #0,-(sp)      ; Drive A = 0
        move      #7,-(sp)      ; Funktion GETBPB
        trap      #13           ; BIOS aufrufen
        addq.l    #4,sp
        move.l    d0,a4         ; Zeiger auf BPB
        move.l    #texte,a5     ; Zeiger auf Texttabelle

        move      #7,d1         ; Schleife 8 mal
```

```
loop    move    (a4)+,d2     ; ein Wert uebergeben
        bsr     print        ; und ausgeben
        dbra    d1,loop      ; u.s.w.

        move.w  #1,-(sp)     ; mit Funktion CONIN
        trap    #1           ; auf Taste warten
        addq.l  #2,sp        ; Stack korrigieren
        clr     -(sp)        ; Mit TERM
        trap    #1           ; zum Desktop

print   move.l  a5,-(sp)     ; String ausgeben
        move    #9,-(sp)     ; mittels PRTLIN-Funktion
        trap    #1
        addq.l  #6,sp
        add.l   #tlen,a5     ; Textzeiger weiter
        move    d2,-(sp)     ; nun den Wert
        pea     buffer       ; wandeln
        bsr     bindec       ; Ergebnis in buffer
        pea     buffer       ; buffer ausgeben
        move    #9,-(sp)     ; mittels PRTLIN-Funktion
        trap    #1
        addq.l  #6,sp
        rts

        include "bindec.s"   ; hier steckt bindec

        data
texte   dc.b    13,10,'Sektorgroesse in Bytes     : ',0
        ds.w    0
tlen    equ     *-texte
        dc.b    13,10,'Sektoren pro Cluster       : ',0
        ds.w    0
        dc.b    13,10,'Cluster-Groesse in Bytes   : ',0
        ds.w    0
        dc.b    13,10,'Sektoren im Directory      : ',0
        ds.w    0
        dc.b    3,10,'Sektoren pro FAT            : ',0
        ds.w    0
        dc.b    13,10,'Start-Sektor von FAT 2     : ',0
        ds.w    0
        dc.b    13,10,'Start-Sektor Daten         : ',0
        ds.w    0
        dc.b    13,10,'Anzahl Daten-Cluster       : ',0
        bss
buffer  ds.b    6
        end
```

Bild 8.4: Ausgabe des BIOS-Parameter-Blocks BPB

Als ersten Parameter erwartet die Funktion die Nummer des Laufwerks. Dabei gilt A = 0, B = 1, C = 2 u.s.w. Versuchen Sie einmal, das Programm so zu erweitern, daß es mit CONIN startet. Wenn Sie von der Zahleneingabe #'0' bzw. von der Buchstabeneingabe #'A' subtrahieren, hätten Sie schon den Parameter. Die Bereichsprüfung, die Unterscheidung von Zahlen und Buchstaben und bei letzteren das Erzwingen von Großbuchstaben (siehe Kapitel 5) wären die nächsten Schritte.

Nun aber zum Ist-Stand: A4 wird als Zeiger durch den BPB, A5 als Zeiger durch die Texttabelle eingesetzt. In der Schleife »loop« wird immer nur »(a5)+« an das Unterprogramm »print« übergeben und dabei automatisch auf den nächsten Wert (Postinkrement) gestellt.

»print« gibt zuerst den Text aus und dann den Wert. Die Wertausgabe erfolgt mit Hilfe der Routine »bindec« aus Kapitel 7. Sie wird hier einfach mit der Include-Anweisung eingezogen. Falls das Ihr Assembler nicht kann, so ist das auch kein Problem. Jeder gute Editor (zum Beispiel First Word [WP aus]) erlaubt Ihnen, den Text von »bindec« in den Quelltext einzufügen. Die Textausgabe gestaltet sich recht einfach, weil ich alle Texte auf die gleiche Länge gebracht habe. Diese Länge wird mit der Anweisung

```
tlen    equ    *-texte
```

der Konstanten »tlen« zugewiesen. Beachten Sie bitte, daß in einem solchen Fall die Länge nach der Anweisung »ds.w 0« (oder EVEN) ermittelt werden muß. Auf jeden Fall reicht es, nach jeder Ausgabe »tlen« auf A5 zu addieren, und schon zeigt A5 auf den nächsten Text.

8.8 FAT und Cluster (und halbe Bytes)

FAT heißt »File Allocation Table«, deutsche Informatiker nennen das Blockzuweisungs-Tabelle. Mit Blöcken sind dabei die Cluster gemeint. Ein Cluster besteht aus 2 Sektoren oder 1024 Bytes und ist die kleinste Einheit, die einem File zugewiesen werden kann.

Die Cluster selbst sind ab relativem Sektor 18 bis zum Ende der Diskette fortlaufend angeordnet. Solange wie möglich versucht das TOS, einem File fortlaufende Clusternummern zuzuordnen. Durch das Löschen oder Verkürzen von Files entstehen jedoch Lücken in dieser Folge. Diese müssen irgendwann wieder gefüllt werden. Wird ein File vergrößert, lassen sich die neuen Cluster zwangsläufig nicht mehr an das Ende der Folge hängen, wenn da schon ein anderes File liegt. Langer Rede kurzer Sinn: Die Cluster eines Files können »wild« über die Diskette verstreut sein (englisch scattered). Deshalb muß es eine Tabelle geben, in der notiert wird, welche Clusternummern zu welchem File gehören. Das wäre die FAT.

Die FAT an sich ist recht einfach konstruiert. Am besten kann man sie sich als einen eindimensionalen Array vorstellen, der genau 1709 Elemente hat. Die erste Clusternummer eines Files, der sogenannte Start-Cluster, steht immer im Directory-Entry. Diese Nummer ist der Tabellenplatz, auf dem die nächste Clusternummer zu finden ist. Auf diesem Platz steht wieder die nächste Cluster-Nummer und das setzt sich so fort, bis in einem Platz $FFFF (4095) angetroffen wird, was »Ende der Liste« heißt.

Ein Beispiel:

Platz	Wert
2	3
3	7
4	5
6	4095
7	8
8	4095

Im Directory-Eintrag von File XXX steht zum Beispiel 2. Dann heißt das, XXX startet mit Cluster 2. Er belegt noch die Cluster 3 (Platz 2 verweist auf 3) und 7. Platz 7 verweist zwar noch auf Platz 8, doch der darf nicht mehr gezählt werden, weil dort eine Endmarke (4095) eingetragen ist.

Soweit die Theorie, doch nun zur Praxis. Die FAT belegt 5 Sektoren auf der Diskette und eine Sicherheitskopie davon (FAT 2) nochmals fünf. Den ersten Sektor einer FAT habe ich nun in Bild 8.5 mittels eines Disk-Monitors sichtbar gemacht. In diesem Beispiel belegt das erste File der Diskette die Cluster 2 (lt. Directory) und 3, 4, 5 und 6 lt. FAT. Mit viel Phantasie kann man diese Zahlen erkennen, aber wo steckt da das System?

```
F7FF FF03 4000 0560 00FF FFFF 09A0 000B E003 0DE0 000F 0001 1120 0113 4001 1560
0117 8001 19A0 011B C001 1DE0 011F 0002 2120 0223 4002 2560 0227 8002 29A0 022B
C002 2DE0 022F 0003 3120 0333 4003 3560 0337 8003 39A0 033B C003 3D30 073F 0004
4120 0443 4004 4560 0447 8004 49A0 044B C004 4DE0 044F 0005 5120 0553 4005 5560
0557 8005 59A0 055B C005 5DE0 055F 0006 6120 0663 4006 6560 0667 8006 69F0 FF6B
C006 6DE0 066F 0007 7120 07FF 4F07 7560 0777 8007 79A0 077B C007 7DF0 0DFF 0F08
8120 0883 4008 8560 0887 8008 89A0 088B C008 8DE0 088F 0009 9120 0993 4009 9560
09FF 8F09 99A0 099B C009 9DE0 099F 000A A120 0AA3 400A A560 0AA7 800A A9A0 0AAB
C00A ADE0 0AAF 000B B120 0BB3 400B B560 0BB7 800B B9A0 0BBB C00B BDE0 0BBF 000C
C120 0CC3 400C C560 0CC7 800C C9A0 0CCB C00C CDE0 0CCF 000D D120 0DD3 400D D560
0DD7 800D D9A0 0DDB C00D DDE0 0D32 910E FFFF FFFF 4F0E E560 0EE7 800E F3A0 0EEB
C00E EDE0 0EEF 000F FF2F 0FFF 400F F5A0 0FF7 800F F9F0 FFFB C00F FFEF 0F03 0110
01F1 FFFF 4F10 FFFF FF07 8110 09A1 10FF CF10 0DE1 100F F1FF 11F1 FFFF 5F11 3061
1117 8111 19A1 111B C111 1DE1 111F 0112 2121 1223 4112 26F1 FF27 8112 29A1 122B
C112 2DE1 122F F1FF 31F1 FFFF FFFF 00F0 FFFF 0F00 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 F0FF 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

Bild 8.5 Die FAT, wie sie der Computer sieht

Erster Hinweis zur Lösung des Rätsels: Auch hier sind die Zahlen im 8088-Format notiert.

Der nächste Trick: Jeder Entry (Platz) belegt 12 Bit, also 1 1/2 Bytes! Jedenfalls gilt das für Disketten, nur bei Festplatten sind es 16 Bit. Die ersten beiden Entries (0 und 1) sind reserviert. Das erste dieser 3 Bytes gibt den Diskettentyp an. Lt. IBM müßte da $FC (einseitige Disk, 9 Sektoren) stehen oder $FD (zweiseitig). Bei Atari findet man aber immer nur $F7. Die beiden nächsten Bytes sind immer $FF. Weil die Entries 0 und 1 reserviert sind, ist die niedrigste Cluster-Nummer 2. Lesen wir die FAT nun ab Entry 2 (zählt ab Null) und teilen sie dabei gleich in 3 Nibbles (1.5 Bytes) ein, so ergibt sich

034 000 056 00F FFF

Daraus müssen aber die Zahlen

003 004 005 006 FFF

entstehen, natürlich per Programm und in Assembler.

Die Lösung bringt Bild 8.6 und damit auch die Auflösung des Rätsels.

```
* FAT1 Cluster-Liste aus FAT lesen

        move    #2,d1       ; Startcluster
        bsr     print       ; ausgeben

loop    lea     fat,a4      ; Zeiger auf FAT
        move    d1,d4       ; Cluster kopieren
        muls    #3,d1       ;  mal 1.5
        asr     #1,d1
        ext.l   d1          ; als Langwort
        add.l   d1,a4       ;  auf Fat-Start addieren
        move.b  1(a4),d1    ; High-Byte in 8088
        asl     #8,d1       ; auf 68000-Position
        move.b  (a4),d1     ; nun Low-Byte hinzu
        btst    #0,d4       ; teste Entry
        beq     gerade      ; wenn gerade
        asr     #4,d1       ; sonst schiebe
gerade  andi    #$FFF,d1    ; nur 12 Bit gueltig
        cmpi    #$FFF,d1    ; Ende der Liste?
        beq     fini        ; wenn so
        bsr     print       ; sonst Cluster drucken
        bra     loop        ; und auf ein Neues

fini    move    #1,-(sp)    ; Funktion CONIN
        trap    #1          ; GEMDOS aufrufen
        addq.l  #2,sp       ; Stack korrigieren
        move    #0,-(sp)    ; Funktion TERM(inate)
        trap    #1
```

```
print   ext.l    d1            ; force integer
        move     d1,-(sp)      ; nun den Wert
        pea      buffer        ; wandeln
        bsr      bindec        ; Ergebnis in buffer
        pea      buffer        ; buffer ausgeben
        move     #9,-(sp)      ; mittels PRTLIN-Funktion
        trap     #1
        addq.l   #6,sp
        rts
        include  "bindec.s"

        data
fat     dc.w     $F7FF,$FF03,$4000,$0560,$00FF,$FFFF
        bss
buffer  ds.b     6
        end
```

Bild 8.6 So hangelt man sich durch die FAT

Der Einfachheit halber habe ich den Beginn der FAT als Konstante definiert, das Lesen von Sektoren wird erst im nächsten Abschnitt behandelt. Die Taktik ist nun folgende:

1. Multipliziere die Clusternummer mit 1.5. Der Grund: 1.5 Bytes sind 12 Bit.
2. Addiere das Produkt auf die Startadresse der FAT, lese ab dieser Adresse ein Wort.
3. Bringe das Wort von 8088- in 68000-Notation
4. In diesem Wort sind nur 12 Bit gültig. Wenn die Clusternummer, mit der wir eingestiegen sind, eine gerade Zahl war, dann sind es die 12 niederwertigen Bits (stehen schon rechtbündig richtig). Andernfalls sind es die 12 höherwertigen Bits, sie sind also um 4 Stellen nach rechts zu schieben.
5. Die 12 Bit sind mittels »Maske« zu selektieren. Ist das Ergebnis $FFF, so sind wir am Ende der Liste, also fertig. Andernfalls geht es weiter bei 1.

Im Listing habe ich den Start-Cluster als Nummer 2 dem Register D1 zugewiesen. Diese Zwei wird mittels der Ihnen schon bekannten »bindec«-Routine ausgegeben. Jetzt wird die Start-Adresse der Tabelle dem Register A4 zugewiesen, und nun hätten wir fast ein Problem. Wie multipliziert man in Assembler mit 1.5? Da Sie wahrscheinlich jetzt nicht 200 Seiten über die sinnvolle Implementation von Fließkomma-Arithmetik lesen wollen, machen wir uns die Sache einfach. Wir multiplizieren mit 3 und dividieren dann durch 2. Letzteres natürlich wieder nur mit einem »asr«. Das Ergebnis wird auf A4 addiert, doch nun wird es trickreich. Mit »move.b 1(a4),d1« wird das zweite Byte, das in 8088-Notation höherwertige, gelesen und anschließend mit »asl« sofort auf den richtigen Platz (als Highbyte im 68000-Format) geschoben. Nun wird mit »move.b (a4),d1« das niederwertige Byte geholt, und das steht dann gleich richtig.

Nächstes Problem: Wir hatten ja festgestellt, daß bei einem ungeraden Einstiegs-Cluster das Wort um 4 Bit nach rechts geschoben werden muß. Wie testet man nun auf gerade oder ungerade? Schauen wir uns dafür ein paar Zahlen im Binär-Format des Rechners an:

0001 = 1
0010 = 2
0011 = 3
0100 = 4

Sie sehen, das niederwertige Bit, auch Bit Null genannt, ist bei ungeraden Zahlen Eins, bei geraden Zahlen Null. Folglich kann man mit dem Bit-Test-Befehl, zum Beispiel als

```
btst    #0,d4
```

diese Frage klären. Alle Bit-Befehle des 68000 testen auch das angesprochene Bit. Das Ergebnis steht immer als Komplement im Z-Flag. War also das Bit 1, ist das Z-Flag 0. War es 0, so ist das Z-Flag 1. Der Zustand Z=1 wird mit BEQ abgefragt, Z=0 hingegen mit BNE. Nun müssen wir nur noch die 12 gültigen Bit mit dem schon bekannten AND maskieren und hätten das Ergebnis. Ist dieses ungleich $FFF, war es ein gültiger Cluster. Dieser wird der neue Einstiegs-Cluster, und weiter geht's wie gehabt.

Nun gibt es zwar eine alte Regel für Programmierer, die da sagt, ein Programm, das läuft, darf nicht mehr angefaßt werden, ich will jedoch aus didaktischen Gründen dagegen verstoßen und die Aufgabe noch einmal anders lösen.

Normalerweise bearbeitet man Tabellen mit dem Befehl

```
move    offset(an,dn),rn
```

hier zum Beispiel mit

```
move    0(a4,d1),d1
```

Darin soll A4 der Start der FAT-Tabelle sein und D1 die Cluster-Nummer. Das geht im ersten Ansatz mit Sicherheit schief. »d1« muß nämlich ein Wort oder Langwort sein und die gesamte (effektive) Adresse dann gerade. Der Cluster hingegen kann 3 oder 5 heißen, was eine ungerade Adresse ergibt. Des Rätsels Lösung bringt Bild 8.7.

```
* FAT2 Cluster-Liste aus FAT lesen

        lea     fat,a4          ; Zeiger auf FAT
        move    #2,d1           ; Startcluster
        bsr     print           ; ausgeben

loop    move    d1,d4
        muls    #3,d1           ; mal 1.5
        asr     #1,d1
        move    d1,d2           ; retten
        bclr    #0,d1           ; auf Wort justieren
        move.l  0(a4,d1),d1     ; aus FAT lesen
```

```
        asr.l   #8,d1       ; 8 Bit immer
        btst    #0,d2       ; wurde auf Wort justiert?
        bne     s8          ; wenn nicht
        asr.l   #8,d1       ; sonst noch 8 Bit schieben
s8      move    d1,d2       ; vertausche low high
        asl     #8,d1
        asr     #8,d2
        move.b  d2,d1       ; nun 68000-Format
        btst    #0,d4       ; teste Entry
        beq     gerade      ; wenn gerade
        asr     #4,d1       ; sonst schiebe
gerade  andi    #$FFF,d1    ; nur 12 Bit gueltig
        cmpi    #$FFF,d1    ; Ende der Liste?
        beq     fini        ; wenn so
        bsr     print       ; sonst Cluster drucken
        bra     loop        ; und auf ein Neues

fini    move    #1,-(sp)    ; Funktion CONIN
        trap    #1          ; GEMDOS aufrufen
        addq.l  #2,sp       ; Stack korrigieren
        move    #0,-(sp)    ; Funktion TERM(inate)
        trap    #1

print   ext.l   d1          ; force integer
        move    d1,-(sp)    ; nun den Wert
        pea     buffer      ; wandeln
        bsr     bindec      ; Ergebnis in buffer
        pea     buffer      ; buffer ausgeben
        move    #9,-(sp)    ; mittels PRTLIN-Funktion
        trap    #1
        addq.l  #6,sp
        rts

        include "bindec.s"

        data
fat     dc.w    $F7FF,$FF03,$4000,$0560,$00FF,$FFFF
        bss
buffer  ds.b    6
        end
```

Bild 8.7 Lesen der FAT mit spezieller Adreß-Arithmetik

Am Anfang deckt sich das Listing mit dem vorherigen bis hin zum Befehl

```
        bclr    #0,d1
```

Mit dieser Bit-Clear-Anweisung kann ein einzelnes Bit gelöscht werden. Praktisch heißt das: Wenn die Adresse gerade war, tue nichts, sonst subtrahiere eins. Damit erzwingt man eine gerade Adresse und zwar die nächst niedere. Manchmal sieht man in den Listings auch

```
        andi    #$FFFE,d1
oder    andi    #-2,d1
```

Das schreiben Leute, die von anderen CPU's her kommen und den Befehlssatz des 68000 noch nicht ganz gelernt haben. Beide Anweisungen maskieren nämlich nur Bit 0 aus. »-2« ist die Kurzschreibweise von »$FFFE«. Der Assembler geht dann nämlich automatisch in die Notation für vorzeichenbehaftete Zahlen (Zweier-Komplement).

Wenn der Cluster ungerade war, sind wir also um ein Byte in der Adresse zu tief. Lesen wir ab dieser Adresse ein Wort, so fehlt uns vom gewünschten Wort ein Byte. Deshalb müssen wir ein Langwort lesen. In diesem Langwort ist dann aber das erste Byte und das letzte zuviel. Das Problem ist einfach zu lösen. Schiebt man nämlich die Bits des Wortes um acht nach rechts, steht das gesuchte Wort »unten«. War hingegen die Adresse gerade, so steht das gesuchte Wort richtig ab dieser Adresse, in unserem Langwort dann jedoch im höherwertigen Teil. Nun müssen wir »unser Wort« um 16 Bit nach rechts schieben. In einem solchen Fall ist es immer falsch, zu sagen

wenn gerade .... schiebe 16
wenn ungerade .... schiebe 8

Das ergäbe zwei Abfragen und einen Sprungbefehl extra. Sinnvoller ist es, immer den »gemeinsamen Nenner« zu suchen, um dann zu sagen

schiebe auf jeden Fall um 8,
wenn gerade schiebe nochmals um 8

Diese Taktik verfolgt auch das Listing. Der BCLR-Befehl sagt uns zwar im Z-Flag, ob das Bit gesetzt war, doch wird das Z-Flag auch von den folgenden Befehlen beeinflußt. Deshalb wird das Offset vorab von D1 in D2 kopiert (gerettet) und dann später mit »btst #0,d2« wieder abgefragt. Der Test von Bit 0 dient der Prüfung auf gerade oder ungerade, wie schon geschildert. Nun folgt die Umsetzung in das 68000-Format und zwar so

```
move    d1,d2
asl     #8,d1
asr     #8,d2
move.b  d2,d1
```

Im Wort in D1 sollen das Low- und das Highbyte getauscht werden. Nehmen wir an, in D1 – und nach der Kopie in D2 – steht

	D1	D2
vorher	ABCD	ABCD
nach asl #8,d1	CDxx	ABCD
nach asr #8,d2	CDxx	xxAB
nach move.b d1,d2	CDAB	

Nach dieser Einlage geht es in Bild 8.7 so weiter, wie schon für Bild 8.6 geschildert.

8.9 Einrichten, Schreiben, Lesen und Kopieren von Dateien

Bisher haben wir die sogenannten Low-Level-Routinen behandelt. Das TOS schaute im BIOS-Parameter-Block (BPB) nach, wo das Directory auf der Diskette zu finden ist. Dort wurde ein Name gefunden und nach noch einem Blick in den BPB (wo beginnt die FAT?) die Cluster-Liste zusammengesucht. Jetzt wollen wir es uns etwas bequemer machen und GEMDOS-Routinen benutzen, die uns den größten Teil der Arbeit abnehmen. Folgende Aufgabe ist gegeben:

Der Anwender arbeitet mit einer RAM-Disk. Nach jedem Systemstart kopiert er eine Gruppe von Files auf die RAM-Disk. Er möchte nun, daß dies nach dem Start automatisch erfolgt. Die Auswahl der zu kopierenden Files muß einem Laien möglich und jederzeit bequem zu ändern sein.

Um beim letzten Punkt zu beginnen: Wir legen fest, es gibt einen Ordner mit dem Namen RDFILE. Alle Files in RDFILE werden auf die RAM-Disk kopiert. Damit ist die Aufgabe für den Programmierer schon klar. Er muß das Unterdirectory RDFILE durchsuchen und jeden dort angetroffenen Namen auf Drive C (die RAM-Disk) kopieren. In Bild 8.8 finden Sie die Lösung.

```
* COPY   Kopierprogramm zu Auto-RAM-Disk
*        ==============================
*        (c) 1986 Markt & Technik
*        by Peter Wollschlaeger
;-----------------------------------------------------------
;macros for GEMDOS-functions:

; ACHTUNG: den \ in den Makros ggf. durch /
; oder ? ersetzen!

create  macro                   ; create new file
        move    #0,-(sp)        ; rw status
        pea     \1              ; Filename
        move    #$3C,-(sp)
        trap    #1
        addq.l  #8,sp
        move    d0,\2           ; save handle
        endm

open    macro                   ; open existing file
        move    #0,-(sp)        ; rw status
        pea     \1              ; Filename
        move    #$3D,-(sp)
        trap    #1
        addq.l  #8,sp
        move    d0,\2           ; save handle
```

```
        endm
close   macro                   ; close file
        move    \1,-(sp)        ; push handle
        move    #$3E,-(sp)
        trap    #1
        addq.l  #4,sp
        endm
read    macro    :read file
        pea     \1              ; adress buffer
        move.l  \2,-(sp)        ; file size to read
        move    \3,-(sp)        ; handle
        move    #$3F,-(sp)
        trap    #1
        add.l   #12,sp
        move.l  d0,\4           ; return size read
        endm
write   macro                   ; write file
        pea     \1              ; adress buffer
        move.   /2,-(sp)        ; file size
        move    \3,-(sp)        ; handle
        move    #$40,-(sp)
        trap    #1
        add.l   #12,sp
        endm
setdta  macro                   ; set disk transf adr
        pea     \1              ; adress dta buffer
        move    #$1A,-(sp)
        trap    #1
        addq.l  #6,sp
        endm
sfirst  macro                   ; search first file
        move    #0,-(sp)        ; atrib = normal
        pea     \1              ; file name
        move    #$4E,-(sp)
        trap    #1
        addq.l  #8,sp
        tst     d0
        endm
snext   macro                   ; search next file
        move    #$4F,-(sp);
        trap    #1
        addq.l  #2,sp
        tst     d0
        endm
```

```
writes  macro                   ; print line
        pea     \1
        move.w  #9,-(sp)
        trap    #1
        addq.l  #6,sp
        endm

ertest  macro                   ; error test
        tst     d0
        bmi     error
        endm
*-----------------------------------------------------------
start   writes  msg             ; main loop
        setdta  dta
        sfirst  source          ; search 1th file
        bne     done            ; if none
        bsr     copy            ; else copy
loop    snext                   ; search next
        bne     done            ; if last
        bsr     copy
        bra     loop            ; else loop

done    clr     -(sp)           ; terminate
        trap    #1
*-----------------------------------------------------------
copy    writes  name
        writes  crlf
        move.l  #name,a1        ; put name in dta
        move.l  #sfile,a2       ; after pathname
        move    #13,d1
lp1     move.b  (a1)+,(a2)+
        dbra    d1,lp1

        open    source,d7       ; read source
        ertest
        read    buffer,size,d7,d1
        ertest
        close   d7
        ertest
        move.b  #'C',c          ; put 'C:\'
                                ; Hier aendern, wenn anderer
                                ; Ziel-Drive **************
        move.b  #':',colon      ; before name
        move.b  #'\',slash      ; ****\muß Backshlash sein****
        create  c,d7            ; create new file
        ertest
        write   buffer,d1,d7 ; and write it
        ertest
```

```
        close    d7
        ertest
        rts                      ; that's all folks
*-----------------------------------------------------------
error   writes   ermsg           ; output error msg
        move     #1,-(sp)        ; wait CONIN
        trap     #1
        addq.l #2,sp
        bra      done
        data
source  dc.b     'a:\rdfile\' ; ****\muß Backshlash sein****
sfile   dc.b     '*.*',0
        dc.b     ' '             ; 10 Blanks
msg     dc.b     27,'E Copying',13,10,0
        ds.w 0
crlf    dc.b     13,10,0
        ds.w 0
ermsg   dc.b     13,10,'Abbruch wegen Disk-I/O-Fehler'
        dc.b     13,10,'Druecke eine Taste ... ',0
        bss
dta     ds.b     26              ; siehe Text
size    ds.b     1
c       ds.b     1
colon   ds.b     1
slash   ds.b     1
name    ds.b     14

buffer  equ      *
        end
```

Bild 8.8 Das Kopierprogramm für die automatische RAM-Disk

Zuerst: Alle Routinen zum File-Handling sind als Makros geschrieben. Das hat für Sie zunächst den Vorteil, daß Sie diese Makros in Ihre Bibliothek übernehmen können, dann bringt diese Technik aber auch sehr viel mehr Klarheit in das eigentliche Programm.

Generell läuft das File-Handling im GEMDOS nicht viel anders, als in Hochsprachen. Daraus ergibt sich auch die Folge

File öffnen
Lesen oder Schreiben
File schließen

Die erste eventuelle Alternative ist diese Unterscheidung:

OPEN	öffnet eine existierende Datei
CREATE	kreiert und öffnet eine neue Datei

Wird CREATE auf eine bereits existierende Datei angewandt, wird sie gelöscht und neu eröffnet. Beide Funktionen werden so aufgerufen:

```
move      #status,-(sp)
pea       Filename
move      #Funktion,-(sp)
trap      #1
addq.l    #8,sp
```

Der Status kann sein 0 = Lesen und Schreiben erlaubt
1 = nur Lesen erlaubt

Als Status wären auch noch 2 und 4 möglich, womit versteckte Dateien erzeugt werden. Beide »Öffne«-Funktionen geben in D0 eine Zahl zwischen 6 und 44 zurück. Diese Zahl heißt Handle (Griff), man übersetzt Handle am besten mit Zugriffsnummer. Alle weiteren Operationen laufen nämlich unter Angabe der Handle. Wird als Handle eine negative Zahl zurückgegeben, heißt das Fehler. Eine Liste der GEMDOS-Fehlermeldungen finden Sie im Anhang. Nun sind die Funktionen READ und WRITE schon einfacher zu erklären, nämlich so

```
pea       Adresse_Puffer
move.l    Anzahl_Bytes,-(sp)
move      handle,-(sp)
move      #Funktion,-(sp)
trap      #1
add.l     #12,sp
```

Man muß die Adresse eines Puffers übergeben, in dem die zu schreibenden Daten stehen bzw. in den die zu lesenden Daten hineingeschrieben werden sollen. Dann erfolgt die Angabe der Anzahl der zu lesenden/zu schreibenden Bytes, danach die Handle und schließlich die Funktionsnummer. Nach der Rückkehr steht in D0 die negative Fehlernummer, was ich Ihnen natürlich nicht wünsche, oder die Anzahl der wirklich gelesenen/geschriebenen Bytes. Damit können Sie zum Beispiel bis zum Ende eines unbekannten Files lesen, wenn Sie einfach eine sehr große Zahl zu lesender Bytes angeben. Die Makros SFIRST und SNEXT kennen Sie schon aus dem Programm »Directory lesen«, und das ist eigentlich auch der Kern dieses Programms. Gelesen werden alle Files aus dem Directory RDFILE von Drive A. So ist auch die Konstante »source« definiert. Nach dem Lesen eines File-Namens steht dieser im DTA-Puffer, und hier wird es trickreich. Definiert wurde:

```
dta       ds.b  26
size      ds.b  1
c         ds.b  1
colon     ds.b  1
slash     ds.b  1
name      ds.b  14
```

Normalerweise steht auf Byte 26 und 29 (ab Null gezählt) die File-Größe. Diesen Bereich mißbrauche ich, um vor den Namen das »c:\« zu schreiben, womit ich den Pfadnamen für den Ziel-File auf Drive C einsetze. Daher auch die Labels » c, colon, slash«. Nun dürfte auch das Konzept klar sein, nämlich

Filenamen aus Directory lesen
Namen hinter Pfadnamen »RDFILE« kopieren
Damit File öffnen
File in »buffer« lesen
Pfad für Drive C vor »Name« eintragen
Create »Name«
Schreiben auf »Name«

Damit der User sieht, daß etwas geschieht, wird noch jeder Name auf dem Schirm angezeigt. Außerdem erfolgt nach jeder File-Operation ein Fehlertest. Spricht dieser Test an, wird eine kleine Meldung ausgegegen und auf eine Taste gewartet. Anschließend wird das Programm einfach abgebrochen. Zum Schluß noch ein Hinweis.

Ich habe den Puffer einfach mit

```
buffer    equ    *
```

definiert. Das heißt, der Puffer soll da beginnen, wo der Location Counter (siehe Kapitel 5) gerade steht. Seine Größe ist nicht definiert. Das funktioniert, weil GEMDOS einem Programm den gesamten Speicher zuweist, wenn man es nicht ausdrücklich anders angibt. Die Funktion TERM (Funktion 0) gibt dann den Speicher wieder zurück. Wie man das anders macht, sprich, nur soviel Speicher reserviert, wie man tatsächlich braucht, wird gleich geschildert.

Vorab nur eines: Man sollte nicht mehr Speicher verbrauchen als nötig, damit auch noch andere Programme gleichzeitig im Speicher sein können.

8.10 Der Overhead

8.10.1 Der Programmkopf

Wenn Sie ein Programm-Icon anklicken, wird das Programm vom TOS in den Speicher geladen und gestartet. So weit so gut, doch woher weiß das TOS zum Beispiel etwas über die Größe der Datenbereiche des Programms (data und bss)?

Bild 8.9 zeigt ein kleines Nonsens-Programm im Assembler-Listing und darunter das, was tatsächlich im Programm-File steht.

```
43F9    0000    0000    lea     x1,a1
45F9    0000    0002    lea     x2,a2
4E75                    rts

                        data
1234                    x1              dc.w    $1234
1234    5678            x2              dc.l    $12345678

                        bss
                        ds.w 3
                        end
-----------------------------------------------------
601A    0000    0010    0000    0008    0000    0008  0000
0000    0000    0000    0000    0000    0000    43F9  0000
                                                  ----------
0010    45F9    0000    0012    4E75    0000    1234 1234
        -------------------------               ----------
5678    0000    0000    0002    0600    0000    0000  0000
----
```

Bild 8.9 Assembler-Listing, und was wirklich im Programm-File steht

Im Hex-Dump des Programm-Files habe ich das unterstrichen, was Assembler-Code erzeugt hat. Den Rest hat der Linker hinzugefügt. Das macht er automatisch. Den Teil vor dem Programmbeginn nennt man Programmkopf oder kurz Header. Der Header hat folgende Einteilung:

Adresse	Inhalt im Beispiel	Bedeutung
$00	601A	bra * + $1A ; Sprung zum Programm
$02	0000 0010	Länge des Programmtexts
$06	0000 0008	Länge Data-Segment
$0A	0000 0008	Länge BSS-Segment
$0E	0000 0000	Länge der Symbol-Tabelle
$12-1B	Nullen	reserviert

Nach dem Programm folgt eventuell eine Symbol-Tabelle. Diese erzeugt der Assembler nur auf Wunsch. Dazu gehört dann auch ein geeigneter symbolischer Debugger. Der Symbol-Tabelle bzw. direkt dem Programm folgt die Verschiebe-Tabelle, auch Relokatiertabelle oder neudeutsch »relecation table« genannt. Das Ding bedarf nun einiger Erklärung.

Das Programm soll an jeder Stelle im Speicher laufen können. Folglich kann der Assembler die Anweisung »lea x1,a1« nicht in eine absolute Adresse umrechnen, sondern er gibt dem Linker nur einen Hinweis. Dieser setzt dann eine absolute Adresse ein, wobei er ab Programm-Start bei Null beginnend zählt. Zählen Sie mit, so finden Sie das Label x1 auf Adresse $10. Mit Sicherheit wird aber das Programm nicht ab Adresse 0 geladen. Folglich muß das »$10« auf die neue Adresse umgerechnet werden. Der Lader kann aber einen Befehl wie »lea x1,a1« nicht erkennen, sondern er braucht eine Hilfe, und das ist die Verschiebe-Tabelle. Das erste Langwort in dieser Tabelle gibt den Abstand vom Programmbeginn bis zur ersten umzurechnenden Adresse im Programm an, genau: bis zum Adreßteil des Befehls. Im Listing steht

```
43F9 0000 0000    lea  x1,a1
```

Zu ändern ist das Langwort ab Adresse 2 (gezählt wird ab Null). Auf die dort stehenden Nullen wird die neue Adresse addiert. Die Zwei finden Sie auch in der Verschiebetabelle, doch nun folgt eine Sechs. In der Verschiebetabelle wird nämlich – um Platz zu sparen – jetzt nur noch in Bytes gezählt. Sprich, die nächste zu verschiebende Adresse beginnt 6 Bytes später als ihr Vorgänger.

Da sich aber in Bytes nur Zahlen bis 255 darstellen lassen, wird bei größeren Abständen als 254 (gerade Zahl) ein Byte mit dem Wert Eins eingesetzt und das so oft, bis der Abstand in 254er-Schritten überbrückt ist.

8.10.2 Die Base-Page

Doch der Lader tut noch mehr. Er richtet zuerst die sogenannte Base-Page ein und packt deren Adresse auf den Stack. Danach ruft er unser Programm als Unterprogramm auf. Folglich findet sich die Adresse bei »4(sp)«. Wenn Sie auf die Base-Page zugreifen wollen, schreiben Sie als erste Anweisung im Programm

```
move.l   4(sp),a0
```

Nun haben Sie Zugriff

mit	auf
0(a0)	Beginn der TPA (Programmspeicherbereich)
4(a0)	Ende der TPA
8(a0)	Beginn des Programms
$C(a0)	Programmlänge (in Bytes)
$10(a0)	Beginn Data-Segment
$14(a0)	Länge Data-Segment
$18(a0)	Beginn BSS
$1C(a0)	Länge BSS
$81(a0)	Adresse Kommandozeile

Bild 8.10 zeigt, wie man an die Kommandozeile herankommt

```
* BASE
        move.l    4(sp),a0      ; Adresse Base Page
        lea       $81(a0),a4    ; Adresse Kommando
loop    move.b    (a4)+,d2      ; hole ein Byte
        move      d2,-(sp)      ; drucke via CONOUT
        move      #2,-(sp)
        trap      #1
        addq.l    #4,sp
        cmpi.b    #13,d2        ; war es CR?
        bne       loop          ; nein, weiter

        move      #1,-(sp)      ; warte auf Taste
        trap      #1
        addq.l    #2,sp
        clr       -(sp)         ; und zum Desktop
        trap      #1
        end
```

Bild 8.10 Lesen der Kommandozeile

Beachten Sie bitte, daß ich nicht mit A0 weiterarbeite, sondern mit A4. Das ist immer empfehlenswert, wenn man TOS-Funktionen aufruft, da diese A0 ändern. Um das Programm laufen lassen zu können, müssen Sie es BASE.TTP nennen oder als »TOS nimmt Parameter« anmelden. TTP heißt übrigens »TOS Takes Parameter«.

Kapitel 9

Schneller geht es nicht: Grafik in Assembler

Befehlsemulatoren des 68000

Line-A-Grafik

Line-A-Variable

Beispiele

9.1 Die Befehlsemulatoren des 68000

Wie schon geschildert, gibt es zahlreiche Möglichkeiten, den 68000 in eine sogenannte Exception zu bringen. Bisher haben wir dieses auch schon kräftig mit »Trap #1« praktiziert. Jetzt möchte ich Ihnen eine Art von Exception vorstellen, die besonders viele Möglichkeiten bietet.

Ein Befehl des 68000 ist immer ein Wort (16 Bit), ggf. gefolgt von weiteren Worten mit den Operanden. Im Befehlswort selbst bezeichnen die 4 höchstwertigen Bits die Befehlsgruppe. Zwei dieser Gruppen sind für spezielle Zwecke reserviert, nämlich für die, deren Befehlsworte mit

```
        1010
und     1111
```

beginnen. In hexadezimaler Schreibweise ist

```
        1010 = $A
und     1111 = $F
```

Sobald der 68000 einen Befehl antrifft, der mit $A oder $F beginnt, läuft er in eine Exception. Zu jeder Exception gehört bekanntlich ein Vektor und diese wären hier (xxx steht für beliebige Werte)

```
        $28  im Falle von $Axxx
und     $2C  im Falle von $Fxxx
```

In diesen Vektoren sollten nun die Adressen der Routinen stehen, die diese Ausnahmen behandeln. Interessant ist nun, daß diese Routinen angesprungen werden, sobald ein Befehl auftaucht, der mit den 4 Bit $A bzw. mit $F beginnt. Die übrigen 12 Bit können eine Nachricht enthalten, praktisch eine Zahl. Bei 12 Bit kann diese Zahl 0 bis 4095 lauten. Die Routine selbst kann nun diese Zahl als einen Befehl auffassen, und wenn man das raffiniert schreibt, kann das zum Beispiel so aussehen:

```
Linie   equ   $A001
Kreis   equ   $A002
u.s.w.
```

Im Programm muß man dann nur noch sagen

```
dc.w    Linie
dc.w    Kreis
```

Man hat damit praktisch neue Befehle geschaffen, man sagt auch emuliert. Daher der Begriff Befehlsemulator.

Nun läuft das Ganze leider nicht automatisch, aber der 68000 unterstützt uns kräftig. Wenn er in eine Exception läuft, dann packt er zuerst folgende Informationen auf den Stack

```
0(a7)  —>  Statusregister
2(a7)  —>  PC beim Auftreten der Exception
```

(Bei den Exceptions Bus-Error und Adreß-Error gibt es noch mehr Informationen).

»a7« ist hier natürlich der Supervisor-Stackpointer, denn automatisch landen wir mit einer Exception im Supervisor-Modus. Nun schauen wir uns einmal an, was der ST macht, wenn er einen Befehl der Art »$Axxx« antrifft. Bild 9.1 zeigt, was ich da mittels eines Disassemblers entdeckt habe.

```
* LINE_A Line-A-Emulator des ST
*-------------------------------------------------------------
      move.l   2(a7),a1            ; PC holen
      move     (a1),d2             ; $Axxx -> d2
      and      #$FFF,d2            ; das "A" ausblenden
      addq.l   #2,a1               ; zeigt nun auf Befehl nach
                                     $Axxx
      move.l   a1,2(a7)            ; wird Return-Adresse
      cmp      #$F,d2              ; > $A00F?
      bhi      exit                ; nur A-Traps 0-15 erlaubt
      lsl      #2,d2               ; Befehls-Nr * 4
      move.l   table(pc,d2),a1     ; Adresse der Routine
      movem.l  d3-d7/a3-a5,-(a7)   ; Register retten
      jsr      (a1)                ; Routine aufrufen
      movem.l  (a7)+,d3-d7/a3-a5   ; Register zurueck
      exit     rte                 ; das war's
*-------------------------------------------------------------
* $A000 folgt direkt
* ------------------
A000  lea      $293A,a0            ; Start Line-A-Variable
      move.l   a0,d0
      lea      $FC9D20,a1          ; Tabelle mit Adressen der
                                     System-Font-Header
      lea      $FC0CE0,a2          ; Tabelle der Adressen der
                                     Line-A-Routinen
      rts
table dc.l     A000
;     dc.l     A001
;     u.s.w.

end
```

Bild 9.1 Der Line-A-Handler des Atari ST

Nehmen wir an, im Programm wurde »dc.w $A007« geschrieben. Dann hat dieses eine Exception ausgelöst, und wir sind beim ersten Befehl des Listing gelandet. Auf dem Stack befindet sich bei a7 plus 2 der PC-Stand im Moment der Exception. Folglich kann man mit

```
move.l   2(a7),a1
```

diesen PC, also praktisch den Zeiger in unser Programm, in das Register A1 kopieren. A1 zeigt jetzt auf unser »$A007«. Nun folgt

```
        move    (a1),d2
```

und damit steht »$A007« in d2. Durch das anschließende

```
        and     #$FFF,d2
```

wird das »A« ausgeblendet und in D2 hätten wir nun »007«, sprich die Funktionsnummer. Bevor damit gearbeitet wird, wird noch der Rückzug vorbereitet.

```
        addq.l  #2,a1
```

stellt A1 auf den Befehl nach dem »$A007« und diese Adresse geht mit

```
        move.l  a1,2(a7)
```

auf den Stack und bleibt dort als zukünftige Return-Adresse.

Den Rest kennen Sie schon aus Kapitel 5. Aus der Funktions-Nummer (hier in D2) und einer Tabelle wird die Adresse der Routine berechnet und diese dann aufgerufen. Zum Schluß schreibt man nicht »RTS« sondern »RTE«, was »Return from Exception« heißt. Vorher wird aber noch das Statusregister vom Stack geholt, daher die Zwei in »move.l a1,2(a7)«.

Unmittelbar auf den Trap-Handler folgt schon die erste Routine, nämlich $A000. Sie trägt den Namen Initialisierung (kurz Init). Praktisch tut sie nichts weiter, als die Adressen in die Register einzutragen, wie im Kommentarfeld beschrieben. Mit dem kleinen Programm laut Bild 9.2 werden diese Adressen dargestellt.

```
* AXXX Adressen der Line-A-Routinen
        dc.w    $A000           ; Init aufrufen
                                ; a2 zeigt nun auf Adress-Ta-
                                  belle
        move    #15,d6          ; 16 Werte ausgeben
loop    move.l  (a2)+,d2

; Start von PRTHEX
        move    #7,d1           ; Langwort hat 8 Nibbles
next    rol.l   #4,d2           ; Hole ein Nibble
        move.l  d2,d3           ; nach d3
        andi.b  #$0f,d3         ; maskiere es
        addi.b  #48,d3          ; '0'..'9' waere jetzt OK
        cmpi.b  #58,d3            ist es >9?
        bcs     out             ; nein, dann Ausgabe
        addi.b  #7,d3           ; sonst muss es 'A'..'F' sein
out     move    d3,-(sp)        ; print via GEMDOS
```

```
        move    #2,-(sp)        ; mit Funktion CONOUT
        trap    #1
        addq.l  #4,sp
        dbra    d1,next         ; Next nibble
        pea     crlf            ; Zeilenvorschub
        move    #9,-(sp)
        trap    #1
        addq.l  #6,sp
        dbra    d6,loop         ; next Adresse

        move    #1,-(sp)        ; CONIN
        trap    #1
        addq.l  #2,sp
        clr     -(sp)           ; und Ende
        trap    #1
crlf    dc.b    13,10,0
        end
```

Bild 9.2 Ausgabe der Adressen der Line-A-Routinen

Die PRTHEX-Routine kennen Sie schon aus Kapitel 5. Nach dem Aufruf von »$A000« zeigt A2 auf die erste Adresse. Folglich kann man mit »move.l (a2)+,d2« jeweils eine Adresse an »PRTHEX« übergeben und das solange, bis der Schleifenzähler (D6) abgelaufen ist.

9.1.1 Line-F-Emulator

Grundsätzlich unterscheidet sich LINE-F von LINE-A nur durch den ersten Buchstaben. Sie könnten sich also selbst einen Befehlsemulator schreiben und dazu als Vorlage die Routine von Bild 9.1 benutzen. Sie müßten dann nur vor dem ersten Aufruf den Exception-Vektor für LINE-F auf Ihren »Trap-Dispatcher« stellen. Dafür gibt es sogar eine BIOS-Funktion (Nummer 5), der Sie lediglich die Adresse Ihrer Routine und die Vektor-Nummer (11) übergeben müssen. Unbedingt empfehlenswert ist das aber nicht, denn es könnten ja noch mehr Leute auf die Idee kommen, so vorzugehen. Theoretisch ist der Vektor für einen mathematischen Co-Prozessor reserviert. Praktisch hat es sich eingebürgert, ihn für Debugger-Programme zu verwenden.

9.2 Die Line-A-Grafik

Beim Macintosh werden alle Routinen des Betriebssystems (über 500) mittels der »A-Traps« aufgerufen. Dieser Weg ist an sich eleganter als der über die Traps 1–15. Der Grund ist einleuchtend. Beim Aufruf über zum Beispiel Trap 1 müssen Sie zumindest schreiben

```
move      #7,-(sp)      ; Funktion 7
trap      #1
addq.l    #2,sp
```

Im Line-A-Emulator reicht hingegen

```
dc.w      $a007
```

Beim Atari ST bedient der Line-A-Emulator nur den Grafik-Kern des Systems. Hierunter versteht man 16 Routinen, die die grafischen Grundfunktionen bereitstellen, aus denen letztendlich alle Bilder, auch Fenster, Drop-Down-Menüs und Ähnliches entstehen. Bild 9.3 bringt einen schnellen Überblick, in der Art »who is who«; wir gehen gleich detaillierter auf einzelne Routinen ein.

Aufruf	Funktion
$a000	Initialisierung
$a001	Einen Bildpunkt setzen
$a002	Farbe eines Bildpunktes feststellen
$a003	Linie zeichnen
$a004	Horizontale Linie zeichnen
$a005	Rechteck zeichnen und füllen
$a006	Polygon zeichnen und zeilenweise füllen
$a007	Bit Block Transfe
$a008	Text Block Transfer
$a009	Maus-Cursor einschalten
$a00a	Maus-Cursor ausschalten
$a00b	Maus-Cursor ändern
$a00c	Sprite löschen
$a00d	Sprite zeichnen
$a00e	Speicherbereich kopieren

Bild 9.3 Die Line-A-Funktionen im Überblick

9.3 Die Line-A-Variablen

Natürlich benötigen die Line-A-Funktionen auch Parameter. Zum Beispiel muß man schon sagen, von wo nach wo eine Linie gezeichnet werden soll. Zu diesem Zweck gibt es die sogenannten Line-A-Variablen. Praktisch ist dieses nichts weiter, als ein reservierter Speicherbereich. Die Init-Funktion ($a000) liefert in D0 und A0 einen Zeiger auf diesen Bereich. Schaut man sich an, was dahinter steckt, so findet man in einem ST mit ROM-TOS auch nur die schlichten Befehle

```
lea       $293A,a0
move.l    a0,d0
```

Wie auch immer, wir wissen nun, wo die Line-A-Variablen starten, und die Line-A-Routinen wissen das natürlich auch. Es bedarf allerdings noch weiteren Vereinbarungen, nämlich, wo in diesem Speicherbereich was steht. Diese Angaben können sich natürlich nur auf die Startadresse beziehen, in 68000-Schreibweise wäre das das Offset zu A0. Hier wieder ein paar Beispiele:

Adresse	Bedeutung
34(a0)	Linienmuster
36(a0)	Schreibmodus
38(a0)	x1 Ausgangspunkt
40(a0)	y1
42(a0)	x2 Endpunkt zum Beispiel einer Linie
44(a0)	y2
46(a0)	Zeiger —> Füllmuster

Da sich die Zahlen schlecht merken lassen, hat man ihnen symbolische Namen gegeben. In Assembler ist daher diese Schreibweise üblich:

```
x1    equ   38
y1    equ   40
x2    equ   42
y2    equ   44
```

Um nun eine Linie von zum Beispiel links oben (x1=0, y1=0) nach rechts unten (x2=639, y2=399) zu zeichnen, kann man schreiben

```
move    #0,x1(a0)
move    #,y1(a0)
move    #639,x2(a0)
move    #399,y2(a0)
dc.w    $A003
```

Da man in einem typischen Grafikprogramm sehr oft Linien zu zeichnen hat, ist das eine Menge Tipparbeit, bedarf also dringend einer Rationalisierung. Unterprogramme lohnen sich da nicht, man muß ja immer wieder die vier (oder mehr) Parameter übergeben, aber Makros sind hiernach geradezu ideal. Definieren wir also einen Makro namens »line« in dieser Art:

```
line    macro
        move    \1,x1(a5)       ; Zeichen einer Linie
        move    \2,y1(a5)       ; von x1/y1 -> x2/y2
        move    \3,x2(a5)
        move    \4,y2(a5)
        dc.w    $A003
        endm
```

Danach reduziert sich der Aufwand auf eine Zeile der Form

```
line    #0,#0,#639,#399
```

Tatsächlich kann man die Line-A-Grafik in Assembler nur mittels Makros sinnvoll handhaben. Da man auch die symbolischen Namen für die Offsets in der Variablen-Tabelle immer benötigt, sollte man sich eine Include-Datei anlegen, in der die Namen und die Makros notiert sind. In Bild 9.4 finden Sie eine solche Datei. Um sie nicht unnötig lang werden zu lassen, wurde sie auf die Teile begrenzt, die wir für die folgenden Beispiele benötigen. Im Anhang finden Sie eine vollständige Aufstellung aller Namen und Line-A-Funktionen. Damit können Sie dann »LINEA.INC« von Fall zu Fall ergänzen.

```
* LINEA.INC Include-File als Basis der Line-A-Grafik

;Offsets in Tabelle der Line-A-Variablen
;---------------------------------------
v_planes equ        0    ; Anzahl Video-Planes
v_lin_wr equ        2    ; Bytes/Video-Zeile
fgbp1    equ       24    ; Die 4 Bit-Planes
fgbp2    equ       26    ;   fuer
fgbp3    equ       28    ;    die
fgbp4    equ       30    ;      Farbe
lstlin   equ       32    ; sollte -1 sein
ln_mask  equ       34    ; Linienmuster
wrt_mode equ       36    ; Schreibmodus
x1       equ       38    ; Ausgangspunkt
y1       equ       40
x2       equ       42    ; Endpunkt
y2       equ       44
patptr   equ       46    ; Zeiger -> Fuellmuster
patmsk   equ       50    ; Maske Fuellmuster
multifil equ       52    ; 0/1: 1/alle Planes fuellen
clip     equ       54    ; 0/1: Clipping aus/an

init     macro
         dc.w    $A000            ; Initialisierung
         move.l  a0,a5            ; Zeiger auf Line-A-Variable
         move    #-1,lstlin(a5)   ; immer -1
         move    #0,clip(a5)      ; Clipping vorerst aus
         endm

color    macro
         move    \1,fgbp1(a5)     ; s/w (ist massgebend)
         move    \2,fgbp2(a5)
         move    \3,fgbp3(a5)
         move    \4,fgbp4(a5)
         endm
```

```
line    macro
        move    \1,x1(a5)       ; Zeichen einer Linie
        move    \2,y1(a5)       ; von x1/y1 -> x2/y2
        move    \3,x2(a5)
        move    \4,y2(a5)
        dc.w    $A003
        endm
rectf   macro
        move    \1,x1(a5)       ; Rechteck
        move    \2,y1(a5)       ; von x1/y1 -> x2/y2
        move    \3,x2(a5)       ; "gefuellt" zeichnen
        move    \4,y2(a5)
        dc.w    $A005
        endm
dsprite macro
        move    \1,d0           ; X-Koordinate setzen
        move    \2,d1           ; und y
        lea     \3,a0           ; zeige auf Sprite
        lea     \4,a2           ; und auf Sicherungsbereich
        dc.w    $a00d           ; zeichne Sprite
        endm
usprite macro
        lea     \1,a2
        dc.w    $a00c           ; Sprite ausschalten
        endm
```

Bild 9.4 Include-File als Basis für Line-A-Grafik

Beachten Sie bitte speziell den Makro Color. Hier wird festgelegt, welche der vier Bit-Planes für die Farbe benutzt werden sollen. Ein Wert von 1 heißt dann »ja«, 0 hingegen »nein«.

Die höchste Auflösung von 640 mal 400 Punkten in Schwarzweiß (monochromer Monitor) benutzt nur »fgbp1« . In diesem Falle werden die Daten in den anderen 3 Planes ignoriert. Bei der mittleren Auflösung werden »fgbp1« und »fgbp2« benutzt, die beiden anderen werden ignoriert. Nur die niedrige Auflösung braucht viermal die Eins.

9.4 Line-A-Grafik in der Praxis

Es folgen nun drei Beispiele, die Ihnen zeigen sollen, daß der Umgang mit der Line-A-Grafik an sich ganz einfach zu programmieren ist. Das haben inzwischen auch die Hersteller verschiedener Hochsprachen entdeckt und deshalb Teile der Line-A- Grafik in ihren »Sprachschatz« integriert. Da es nun zum Beispiel der Routine »gefülltes Rechteck« sehr egal ist, ob sie von Basic oder Assembler aus aufgerufen wird, ergeben sich praktisch keine Zeitunterschiede, wenn man nur beobachtet, daß in beiden Fällen das Rechteck blitzschnell auf den Schirm geworfen wird. Erst wenn man Line-A-Routinen vielfach hintereinander aufruft, merkt man die Unterschiede, denn die Hochsprachen müssen natürlich einigen Aufwand für die Umsetzung treiben. Diese Fälle sind immer bei Animation (schnelle Bewegungen von Bildern) gegeben. Hier ist dann Assembler eindeutig im Vorteil.

9.4.1 Beispiel 1: Zeichnen von Linien

Das kleine Programm in Bild 9.5 soll vier Linien zeichnen, die ein großes Kreuz und ein großes X über die volle Schirmgröße bilden sollen.

```
* LINE1 ein paar Linien zeichnen

        include "linea.inc"

        init                        ; Line-A initialisieren
        color   #1,#1,#1,#1         ; hochaufloesend
        move    #0,wrt_mode(a5)     ; Replace-Modus

;Zeichne grosses Kreuz und X
;---------------------------
        move    #-1,ln_mask(a5)     ; durchgezogene Linie
        line    #0,#200,#639,#200   ; wird -
        line    #320,#0,#320,#399   ;     |
        line    #0,#0,#639,#399     ;     \
        line    #0,#399,#639,#0     ;     /

        move    #1,-(sp)            ; Funktion CONIN
        trap    #1                  ; GEMDOS aufrufen
        addq.l  #2,sp               ; Stack korrigieren
        move    #0,-(sp)            ; Funktion TERM(inate)
        trap    #1

        end
```

Bild 9.5 Das Zeichnen von Linien

Der erste Befehl legt die Farbe (hier monochrom) fest, dann folgt der Schreib-Modus. Null bedeutet, der neue Bildpunkt überschreibt den alten.

Der nächste Befehl legt das Linienmuster fest. »-1« heißt in hex »$FFFF« oder in binär %1111111111111111. In diesem Falle sind alle Bits 1, die Linie wird durchgezogen. Durch Wahl des Bitmusters können Sie beliebige Linienmuster generieren. Nun folgen 4 Makro-Aufrufe, mit denen die Linien an sich gezeichnet werden. So einfach ist das!

9.4.2 Beispiel 2: Zeichnen von Linien mit Trick

Nun soll eine Grafik wie die in Bild 9.6 erzeugt werden. Sieht doch »mächtig gewaltig« aus. Dahinter steckt jedoch nur sehr wenig Programm, aber ein Trick. Die Lösung zeigt Bild 9.7.

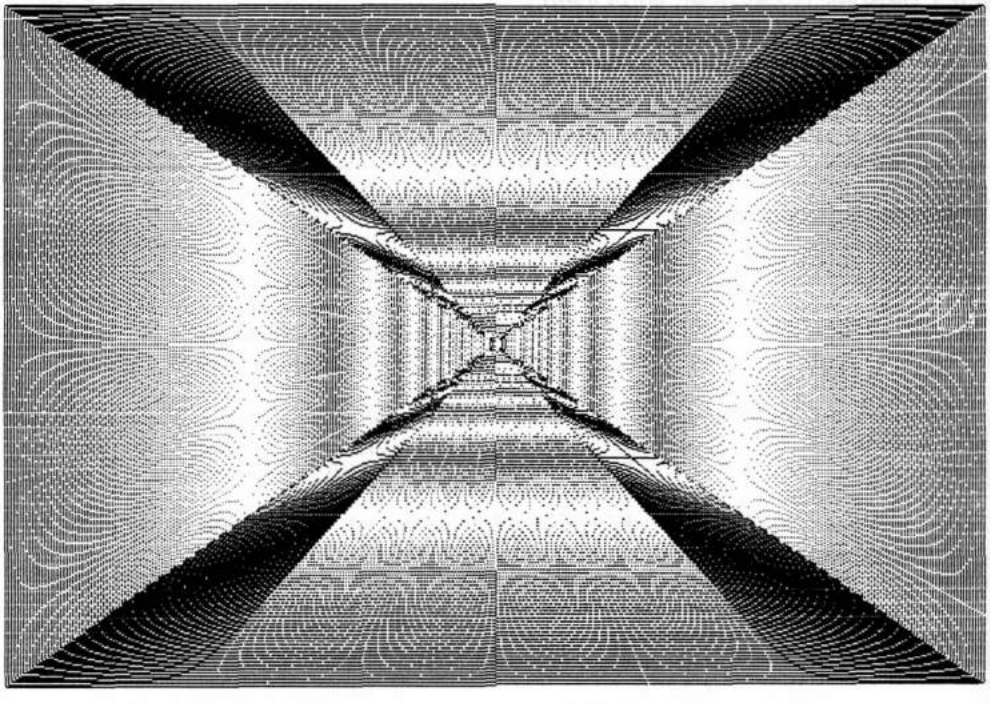


Bild 9.6 Viel Grafik für wenig Programm

```
* LINE2 Grafik-Demo
        include   "linea.inc"

        init                      ; Initialisierung
        color     #1,#1,#1,#1     ; hochaufloesend
        move      #2,wrt_mode(a5) ; Invertier-Modus
        move      #-1,ln_mask(a5) ; durchgezogene Linie

        move      #639,d3         ; Schauen Sie sich das an,
        move      d3,d4
loop1   sub       d3,d4
        line      d3,#0,d4,#399
        move      #639,d4
        dbra      d3,loop1

        move      #399,d3         ; und probieren Sie es aus!
        move      d3,d4
loop2   sub       d3,d4
        line      #0,d4,#639,d3
        move      #399,d4
        dbra      d3,loop2

        move      #1,-(sp)        ; Warte auf Taste
        trap      #1              ;
        addq.l    #2,sp           ;
        move      #0,-(sp)        ; Funktion TERM(inate)
        trap      #1

        end
```

Bild 9.7 Dieses Programm malt Bild 9.6

Die Sache ist auf den ersten Blick schwer zu durchschauen. Dehalb hier die erste Schleife in einer Hochsprache

```
for i=639 to 0 step -1
  line(i, 0, 639-i, 399)
next i
```

Die Schleife läßt sich bequem mit »DBcc« realisieren. Ein kleines Problem ist nur der Ausdruck »639-i«. Daher wird D4 immer wieder mit 639 geladen und dann davon D3 (entspricht dem i) subtrahiert. Der Trick an der ganzen Geschichte ist nun, das der Schreibmodus auf den Wert 2 gesetzt wurde. Zwei bedeutet XOR-Modus, die beiden Pixels (eines auf dem Schirm und das neue) werden nach den Regeln von XOR (exklusiv oder) verknüpft. XOR ist aber die Funktion, die mit »x xor 1« das Bit x kippt, also aus Null

Eins macht, aus Eins Null. So gesehen, heißt XOR nichts weiter als »invertiere«. Hier alle erlaubten Schreibmodi:

0 = Überschreiben
1 = Transparent
2 = Invers
3 = Transparent invers

Experimentieren Sie selbst ein wenig. Das Bild läßt sich durch Wahl des Schreibmodus und des Linienmusters ändern und ergibt jedesmal eine neue, sehr effektvolle Grafik. Gut macht sich auch, wenn Sie über die beiden Schleifen noch eine DBcc-Schleife legen, in der dann zum Beispiel der Schreibmodus variiert wird.

9.4.3 Beispiel 3: Animation

Zum Abschluß dieses Kapitels möchte ich Ihnen die Grundlage zahlreicher Spiele demonstrieren. Der Kern des Listing stammt aus dem Handbuch des Metacomco-Assemblers und wird mit freundlicher Genehmigung von Metacomco wiedergegeben. Falls Sie nur noch sehr wenig Ähnlichkeit mit dem Original feststellen sollten, so liegt dieses an der Anpassung an unser Include-File. Neu ist auch die Abbruchmöglichkeit mittels der Control-Taste. Die Metacomco-Lösung (Abruch nur durch Reset) war mir doch etwas zu brutal.

Das Programm läßt einen Ball auf dem Schirm rollen, der immer, wenn er gegen die Bande läuft, richtig, sprich Ausfallwinkel gleich Einfallwinkel, abprallt. Dahinter stecken nun keine komplizierten trigonometrischen Funktionen, sondern nur schlichte Additionen. Bild 9.8 veranschaulicht das Prinzip.

Nehmen wir an, der Ball soll vom Punkt A zum Punkt B rollen, dort an der Bande abprallen und zum Punkt C rollen. Dazu wird der Ball im Punkt A gezeichnet. Nun wird X und Y um ein paar Punkte inkrementiert und der Ball auf die neue Postion bewegt, was praktisch heißt: lösche an alter Position, zeichne auf neuer. Dieser Vorgang wird nun solange wiederholt, bis der Ball an die Bande stößt, sprich Y gleich Y-Min ist (mitgezählt). Nun muß man doch nur dy negieren (dy = -dy) und dann dx bzw. dy auf das aktuelle X bzw. Y addieren. Sinngemäßes, dann mit anderen Vorzeichen, passiert an den anderen Banden.

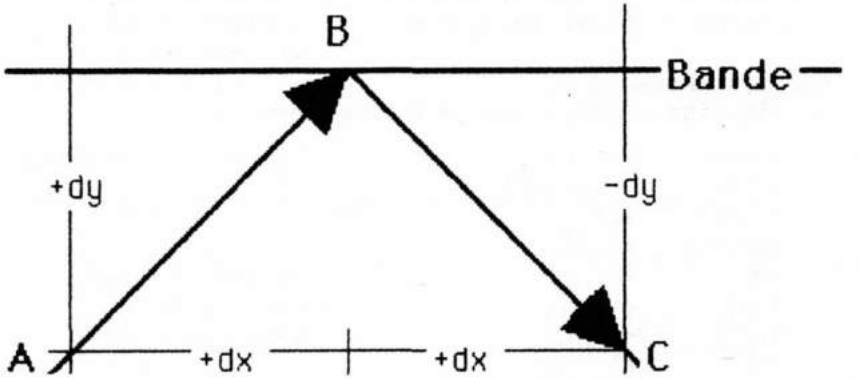


Bild 9.8 Prinzip der Umkehr beim Anstoß an die Bande

Mit diesem Wissen ausgerüstet, können wir uns dem Listing von Bild 9.9 zuwenden.

```
* BOUNCE          The Bouncing Sprite
*****************************************************
*  Die Bounce-Routine stammt von Metacomco und wird *
*  mit freundlicher Genehmigung von Pamela Clare -  *
*  Director Metacomco - verwendet.                  *
*  Das Original ist etwas schwer zu erkennen, weil  *
*  ich es an meine LINE-A-Lib angepasst habe.       *
*  Peter Wollschlaeger.                             *
*****************************************************

        include "linea.inc"

*hier geht es los
*----------------
xmin    equ     8                       ; Setze das Bounce-Window
ymin    equ     8                       ; 8 Pixels vom Rand
xmax    equ     631
ymax    equ     391

start   init                            ; Line-A initialisieren
        color   #1,#1,#1,#1             ; hochaufloesend
        move    #0,wrt_mode(a5)         ; Replace-Modus
```

```
        move.l  #filpat,patptr(a0) ; Zeige auf Fuellmuster
        move    #0,patmsk(a0)      ; keine Fuellmaske
        move    #0,multifil(a0)    ; nur eine Plane
        rectf   #1,#1,#638,#398    ; Schirm schwarz mit mit
                                     weissem Rand
;zeichne Sprite
        dsprite xpos,ypos,sprdef,sprsave

movx    move    #-1,-(sp)          ; teste Sondertasten
        move    #11,-(sp)
        trap    #13
        addq.l  #4,sp
        btst    #2,d0              ; Control-Taste?
        beq     weiter             ; wenn nicht
        clr     -(sp)              ; sonst terminate
        trap    #1
weiter
        move    xpos,d0            ; hole X-Position
        add     xvel,d0            ; addiere Tempo (= Weg)
        cmp     #xmax,d0           ; am Anschlag?
        bgt     xbounce            ; wenn nicht "bounce"
        cmp     #xmin,d0           ; anderer Anschlag?
        bge     noxb               ; Bounce nicht in x

;Bouncing in X-Richtung
;----------------------
xbounce
        neg     xvel               ; Richtung umkehren
        bra     movx               ; neuer Versuch
noxb    move    d0,xpos

movy    move    ypos,d1            ; wie oben bei x
        add     yvel,d1
        cmp     #ymax,d1
        bgt     ybounce
        cmp     #ymin,d1
        bge     noyb

;Bounce in Y-Richtung
;--------------------
ybounce
        neg     yvel
        bra     movy               ; wie bei x

noyb    move    d1,ypos
        usprite sprsave
        dsprite xpos,ypos,sprdef,sprsave

move    #37,-(sp)                  ; Warte auf Austastluecke
        trap    #14
```

```
        addq.l  #2,sp
        bra     movx                    ; und starte wieder
;-----------------------------------------------------------------
xpos    dc.w    100                     ; hiermit fangen wir an
ypos    dc.w    100
xvel    dc.w    3                       ; je groesser
yvel    dc.w    3                       ; desto schneller

        data

;hier sind die Daten fuer den Sprite (happy face)
sprdef
        dc.w    8                       ; X-Offset vom Hot Spot
        dc.w    8                       ; Y
        dc.w    -1                      ; XOR-Sprite-Format
        dc.w    0                       ; Hintergrundfarbe
                                          (keine)
        dc.w    1                       ; Vordergrund schwarz

sprite
        dc.w    %0000000000000000       ; Hintergrund Zeile 0
        dc.w    %0000001111000000       ; Vordergrund Zeile 0
        dc.w    %0000000000000000       ; Hintergrund 1
        dc.w    %0000111111110000       ; Vordergrund 1
        dc.w    %0000000000000000       ; 2
        dc.w    %0001111111111000
        dc.w    %0000000000000000       ; 3
        dc.w    %0011111111111100
        dc.w    %0000000000000000       ; 4
        dc.w    %0111111111111110
        dc.w    %0000000000000000       ; 5
        dc.w    %0111111111111110
        dc.w    %0000000000000000       ; 6
        dc.w    %1111011111001111
        dc.w    %0000000000000000       ; 7
        dc.w    %1111001111001111
        dc.w    %0000000000000000       ; 8
        dc.w    %1111111111111111
        dc.w    %0000000000000000       ; 9
        dc.w    %1111111111111111
        dc.w    %0000000000000000       ; 10
        dc.w    %0111111111111110
        dc.w    %0000000000000000       ; 11
        dc.w    %0111101111011110
        dc.w    %0000000000000000       ; 12
        dc.w    %0011110001111000
        dc.w    %0000000000000000       ; 13
        dc.w    %0011111111111100
```

```
        dc.w    %0000000000000000  ; 14
        dc.w    %0000111111110000
        dc.w    %0000000000000000  ; 15
        dc.w    %0000001111000000
filpat  dc.w    -1                 ; Fuellmuster schwarz
sprsave ds.b    74                 ; Speicher fuer Sprite
        end
```

Bild 9.9 » The Bouncing Sprite«

Das Programm zeichnet nach der Initialisierung mittels des Makros »rectf« ein Rechteck über die volle Schirmgröße mit dem Füllmuster »schwarz«. Das ergibt einen völlig schwarzen Schirm mit einem weißen Rand. Das »Bounce Window« wird lt. den EQU-Direktiven 8 Pixels vom Rand weggelegt, so daß der Sprite visuell nie über diesen Rand hinausläuft. Sprite wäre nun das Stichwort. Beim ST ist ein Sprite ein Muster von 16 mal 16 Bildpunkten oder auch 16 Worten. Damit ist aber nur der Vordergrund definiert. Nochmals 16 Worte bilden den Hintergrund. Leider werden aber nicht Vordergrund- und Hintergrund-Sprite getrennt definiert, sondern immer abwechselnd eine Zeile Hintergrund, eine Zeile Vordergrund. Wenn man das so wie am Ende des Listings gezeigt eintippen muß, gehört schon einiges an Fantasie dazu, sich das Bild richtig vorzustellen. Praktisch ist das aber kein Hinderungsgrund, denn es gibt inzwischen zahlreiche kleine Programme, mit denen man einen Sprite mit der Maus zeichnen kann (Sprite-Editoren), die die Masken generieren.

Zum Sprite gehören dann immer noch die Definitionen, wie unter »spritedef« gezeigt, sowie ein Puffer von 74 Byte Größe (264, wenn Farbe), in dem der Sprite den von ihm überschriebenen Bildschirmbereich sichert. Der Rest ist simpel. Man übergibt die Schirmkoordinaten sowie die Adressen der Spritedefinition und des Puffers und schreibt dann »dc.w $a00d«. Damit wird der Sprite gezeichnet. Noch einfacher ist »unsprite«, das nur die Adresse der Sprite-Definition wissen will, und schon ist der Sprite (vorläufig) vom Schirm verschwunden. Die Hauptschleife des Programms beginnt bei »movx«. Hier teste ich zuerst, ob der Anwender die Control-Taste betätigt hat. Das nur, um einmal diese BIOS-Funktion vorzuführen. Sonst könnte man noch den Tastatur-Status oder die Maus abfragen, nur so etwas wie CONIN (warte auf Taste) darf da natürlich nicht stehen.

Die Geschwindigkeit ist vom Weg abhängig, den der Sprite jeweils zurücklegt. Hier ist der Weg in xvel und yvel definiert und mit 3 Bildpunkten recht klein. Damit ergibt sich ein recht angenehmes Tempo. Bei »Tempo 10« rast der Sprite schon. Da nur Übung den Meister macht, möchte ich Ihnen folgende Aufgabe stellen:

Fragen Sie mit der BIOS-Funktion 11 nicht nur die Control-Taste ab (Bit 2), sondern auch die linke und die rechte Shift-Taste (Bit 0 und Bit 1). Bei »rechts« erhöhen Sie das Tempo, bei links erniedrigen Sie es. Im ersten Ansatz erscheint diese Aufgabe sehr einfach, muß man doch nur in Abhängigkeit vom Testergebnis xvel und yvel um Eins erhöhen oder erniedrigen. Vielleicht fangen Sie sogar so an. Wenn Sie dann zu etwas seltsamen Ergeb-

nissen kommen, nämlich beide Tasten reagieren zwar, aber anders als Sie dachten, dann hätten Sie noch ein Problem zu lösen. Der Rechner fragt die Tastatur mit hoher Geschwindigkeit ab. Sie müßten also, um nur einmal zu inkrementieren, die Tasten für einen Sekundenbruchteil betätigen, was praktisch unmöglich ist. Eine Lösung wäre, einen Zähler einzuführen. Wenn die Shift-Taste betätigt wird, zählen Sie diesen Zähler hoch. Wenn dieser einen gewissen Wert erreicht hat, inkrementieren (dekrementieren) Sie xvel und yvel und setzen den Zähler wieder auf Null.

Kapitel 10

Die Befehle des 68000 im Überblick

In diesem Kapitel soll der Befehlssatz des 68000 im Überblick vorgestellt werden. Zu jedem einzelnen Befehl finden Sie die syntaktischen Formen und die erlaubten Adressierungsarten im Anhang A1. Dort ist auch beschrieben, welche Operandenlängen (Byte, Word, Long) jeweils zulässig sind. Hier geht es primär um die Thematik »was gibt es, und wofür braucht man es«.

10.1 Transfer-Befehle

Befehl	Bedeutung
EXG	Austausch von Registerinhalten
LEA	Laden eines Registers
LINK	Lokalen Datenbereich aufbauen
MOVE	Übertragen (kopieren) von Daten
MOVEA	Übertragen (kopieren) von Adressen
MOVEM	Übertragen (kopieren) mehrerer Register
MOVEP	Übertragen (kopieren) von Daten zur Peripherie
MOVEQ	Übertragen (kopieren) von Konstanten »Quick«
PEA	Adresse auf den Stack bringen
SWAP	Vertauschen der Worte eines Registers
UNLK	Abbau des lokalen Datenbereichs (siehe LINK)

Auffallend sind sicherlich die vielen Varianten des MOVE-Befehls. Hier sollten Sie einmal Ihren Assembler testen. Gute Assembler akzeptieren auch ein MOVE, wo man eigentlich MOVEA hätte schreiben müssen. Sehr gute Assembler geben sogenannte Warnings aus, wenn Sie nicht optimal programmieren, hier also anstatt eines zulässigen MOVEQ nur ein einfaches MOVE schreiben würden.

10.1.1 LINK und UNLK

Besonders erwähnenswert sind sicherlich die Befehle LINK und UNLK (Unlink). Mit diesen Befehlen ist der 68000 besonders gut auf die Aufgabenstellung von Hochsprachen-Compilern vorbereitet. Hier ergibt sich immer das Problem, daß in Prozeduren und Funktionen lokale Variable geschaffen werden müssen, die nur solange existieren sollen, wie die Prozedur (Funktion) aktiv ist. Typischerweise werden deshalb solche Variablen auf dem Stack abgelegt.

Ideal ist es nun, wenn man mit nur einem Befehl den passenden Stackbereich reservieren und dann später auf genauso einfache Art wieder freigeben kann. Genau das bieten die Befehle LINK und UNLK. Wenn nun eine Prozedur (Unterprogramm) eine weitere Prozedur aufruft und diese dann eine dritte usw. und jede dieser Prozeduren mit dem eigenen lokalen

Stack arbeitet, dann entsteht sozusagen eine verkettete Liste, englisch »linked list«. Daher rühren auch die Namen LINK und UNLK.

Schauen wir uns jetzt einmal an, wie das funktioniert. Die Syntax des Befehls lautet

```
            LINK An,#Adreßdistanz
Ein Beispiel:  LINK A6,#30
```

In diesem Fall wird zuerst A6 auf dem Stack abgelegt, praktisch der Befehl

```
        move.l   a6,-(sp)  (Schritt 1)
```

ausgeführt. Nun wird der Stackpointer in das soeben gerettete Register kopiert, sprich

```
        move.l   sp,a6     (Schritt 2)
```

Zuletzt wird die Adreßdistanz auf den Stackpointer addiert, das heißt

```
        add.l    #Adr_Dist,sp
```

Damit hätten wir den lokalen Stack für ein Unterprogramm. Üblicherweise wählt man die Adreßdistanz negativ, da der Stack bekanntlich zu fallenden Adressen hin wächst. Mit UNLK wird der ursprüngliche Zustand wiederhergestellt. Praktisch wirkt UNLK wie

```
        move.l   a6,a7
        move.l   (sp)+,a6
```

Für das Hauptprogramm oder allgemein das aufrufende Unterprogramm hat somit das Adreßregister und der Stackpointer wieder seinen ursprünglichen Wert.

Zu beachten ist noch, daß das Adreßregister nach dem LINK-Befehl eine Kopie des Stackpointers enthält. Somit kann das Unterprogramm sehr einfach auf Daten des aufrufenden Programms zugreifen, wenn dieses die Daten vorher auf den Stack gepackt hat.

10.2 Arithmetische Befehle

Befehl	Bedeutung
ADD	Addition von Daten
ADDA	Addition von Adressen
ADDI	Addition einer Konstanten
ADDQ	Addition einer Konstanten »Quick«
ADDX	Addition mit Übertrag-Bit
CLR	Löschen eines Operanden
CMP	Vergleich zweier Daten
CMPA	Vergleich zweier Adressen
CMPI	Vergleich mit einer Konstanten
CMPM	Vergleich zweier Daten im Speicher

DIVS	Division mit Vorzeichen
DIVU	Division ohne Vorzeichen
EXT	Vorzeichenrichtige Erweiterung
MULS	Multiplikation mit Vorzeichen
MULU	Multiplikation ohne Vorzeichen
NEG	Negation
NEGX	Negation mit X-Bit
SUB	Subtraktion von Daten
SUBA	Subtraktion von Adressen
SUBI	Subtraktion einer Konstanten
SUBQ	Subtraktion einer Konstanten »Quick«
SUBX	Subtraktion mit X-Bit (Borgen)
TST	Teste Operanden gegen Null
ABCD	Addition von BCD-Zahlen
NBCD	Negation von BCD-Zahlen
SBCD	Subtraktion von BCD-Zahlen

Auch hier können gute Assembler wieder glänzen. Zumindest sollten sie mit ADD einverstanden sein, wenn eigentlich ADDA oder ADDI erforderlich ist. Sinngemäßes gilt für CMP und SUB.

Besonders zu loben ist hier der 68000 wegen zweier Eigenschaften. Zuerst: Die arithmetischen Operationen sind auf einer Breite von 32 Bit möglich. In diesem Sinne ist also der 68000 ein echter »32-Bitter«. Sein Datenbus ist zwar nur 16 Bit breit, was dazu führt, daß Langworte in »zwei Portionen« transportiert werden, das interessiert aber nur sekundär. Primär ist wichtig, daß damit mathematische Operationen wesentlich einfacher zu programmieren sind als auf CPU's, die nur 16 oder gar nur 8 Bit breite Operanden zulassen. Sekundär sollte man natürlich darauf achten, daß die Daten solange wie möglich in Registern gehalten werden, denn dann entfällt auch der relativ zeitaufwendige Transfer über den Datenbus. Bei der hohen Geschwindigkeit des 68000 sollte man das allerdings nicht überbewerten. Nur in sehr rechenintensiven Routinen könnte damit etwas erreicht werden.

10.2.1 BCD-Arithmetik

Die BCD-Arithmetik des 68000 wird jeder schätzen, der schon einmal auf einer anderen CPU so etwas programmieren mußte. Eine BCD-Ziffer steht immer in einem Halbbyte (4 Bit). Da sich damit bekanntlich die Zahlen 0 bis 15 darstellen lassen, hier aber nur 0 bis 9 gültig sind, gibt es beim Überlauf einige Probleme. Bei anderen CPU's muß man diesen Fall mit Hilfe des sogenannten Half-Carry-Flags testen, hier kann man einfach addieren. Da als Operandengröße immer nur Byte zugelassen ist, gibt es einen Überlauf bei Überschreiten der Zahl 99. Dieser geht aber automatisch in das X-Flag und wird auch automatisch bei weiteren Additionen mitaddiert. Hier ein Beispiel für die Addition zweier sechsstelliger Zahlen in je 3 Byte.

Zahl	Wert	im Speicher auf Adressen
1	123456	12 auf 1001, 34 auf 1002, 45 auf 1003
2	654321	65 auf 2001, 43 auf 2002, 21 auf 1003

Wie üblich, muß man rechts (bei den Einern) mit der Addition beginnen. Daher ist als Speicheradressierungsart hier auch nur »ARI mit Predekrement« erlaubt. Das Programm sähe dann so aus

```
move    #1004,a1
move    #2004,a2
move    #4,CCR

ABCD    -(a1),-(a2)
ABCD    -(a1),-(a2)
ABCD    -(a1),-(a2)
```

Zu beachten ist dabei dreierlei:

1. Wegen des Predekrements müssen die Zeiger A1 und A2 um Eins größer sein als die tatsächliche Adresse der BCD-Zahlen.
2. Um nicht beim ersten Mal ein zufälliges X-Flag mitzuaddieren, muß man es löschen.
3. Um ein Null-Ergebnis erkennen zu können, sollte man vorher das Z-Flag setzen.

Die Punkte 2 und 3 lassen sich mit der Anweisung »move #4,CCR« sehr einfach zusammen erledigen. Eines bleibt Ihnen allerdings nicht erspart. Sie müssen schon garantieren, daß die Zahlen BCD-Zahlen sind (Abfrage beim Laden). Größere Werte als 9 werden nämlich schlicht falsch addiert.

10.3 Logische Befehle

Befehl	Bedeutung
AND	Logisch UND
ANDI	Logisch UND mit einer Konstanten
EOR	Logisch XOR
EORI	Logisch XOR mit einer Konstanten
NOT	Logisch NICHT (Einerkomplement)
OR	Logisch ODER
ORI	Logisch ODER mit einer Konstanten

Zu diesen Befehlen könnte man vielleicht nur noch anmerken (Sie wissen es schon), daß sie bitweise wirken.

10.4 Bit-Befehle

Befehl	Bedeutung
BCHG	Ändere (kippe) ein Bit
BCLR	Lösche ein Bit
BSET	Setze ein Bit
BTST	Prüfe ein Bit
TAS	Teste und setze Bit 7 eines Byte-Operanden

Die Bit-Befehle halten immer den vorherigen Zustand im Z-Flag fest, führen also 2 Operationen aus. Diese Eigenschaft, und ganz besonders die Fähigkeit von TAS, sind ein gutes Beispiel für eine besondere Fähigkeit des 68000, nämlich die Unterstützung von Multi-Tasking. Mit dem TAS-Befehl wird Bit 7 eines Byte-Operanden im Speicher abgefragt und das Ergebnis wie üblich im Z-Flag notiert. Dann wird eine 1 in Bit 7 geschrieben. Die ganze Folge, also Lesen des Operanden, Abfrage und Zurückschreiben, ist unteilbar, sprich, kann nicht durch einen Interrupt unterbrochen werden. Solche unteilbaren Befehle sind in einem Multi-Tasking-System sehr wichtig. Hier wären zum Beispiel Aufgaben wie Prozeßumschaltung, Synchronisation von Prozessen und ähnliches zu nennen.

10.5 Schiebe- und Rotierbefehle

Befehl	Bedeutung
ASL	Arithmetische Verschiebung nach links
ASR	Arithmetische Verschiebung nach rechts
LSL	Logische Verschiebung nach links
LSR	Logische Verschiebung nach rechts
ROL	Rotieren links herum
ROR	Rotieren rechts herum
ROXL	Rotieren mit X-Bit links
ROXR	Rotieren mit X-Bits rechts

Anzumerken ist hierzu, daß mit einem Befehl um bis zu 31 Bit geschoben/rotiert werden kann. Andere CPU's schaffen mit einer Anweisung immer nur ein Bit. Zwischen ASL und LSL besteht praktisch kein Unterschied, wohl aber einer zwischen ASR und LSR, wie das folgende Bild zeigt

```
ASR     +----->|   Operand   |----+----->  C
        |      |                  |
        +---------+                +---->   X

LSR     0----->|   Operand   |----+----->  C
                                   |
                                   +---->   X
```

Die Zeichnung soll zeigen, daß beim ASR das Vorzeichen-Bit immer wiederhergestellt, praktisch also nicht geschoben wird. Beim LSR wird hingegen ein Null-Bit nachgeschoben. Beim Linksschieben wird sowohl beim ASL als auch beim LSL ein Null-Bit rechts eingespeist.

10.6 Programmsteuer-Befehle

Befehl	Bedeutung
Bcc	Verzweige bedingt
BRA	Verzweige immer
BSR	Verzweige zu einem Unterprogramm
CHK	Checke Datenregister gegen die untere Grenze 0 und eine obere Grenze
DBcc	Bedingte Schleife
JMP	Sprung zu einer Adresse
JSR	Sprung zu einem Unterprogramm
NOP	Keine Operation
RESET	Rücksetzen der Peripherie
RTE	Rückkehr von einer Exception
RTR	Rückkehr mit Laden der Flags
RTS	Rückkehr aus einem Unterprogramm
Scc	Setze ein Byte bedingt
STOP	Halte Programm an
TRAP	Gehe in Exception
TRAPV	Gehe in Exception, wenn V-Flag gesetzt

Zu beachten wäre hier der Unterschied zwischen den Verzweigungs- und den Sprungbefehlen. Erstere sind immer relativ zum aktuellen PC, allerdings auf einen Adreßbereich von +-32 Kbyte begrenzt. Die Sprungbefehle JMP und JSR reichen über den vollen Adreßbereich von 16 MByte.

CHK testet ein Datenregister gegen zwei Grenzen, nämlich 0 und eine im Operanden angegebene Grenze. Spricht der Test an, wird eine Exception ausgelöst. Damit läßt sich sehr einfach eine Bereichsprüfung, zum Beispiel von Array-Indizes, realisieren.

Wenn Sie nun richtig mitgezählt haben, waren das 56 Befehle. Im Vergleich mit anderen CPU's ist das relativ wenig, doch das täuscht. Die meisten der 12 Grundadressierungsarten können auf den Quell- und den Zieloperanden angewendet werden, womit sich über 1000 Varianten ergeben. Andere CPU's geben einigen dieser Varianten eigene Befehlsnamen, womit zwar nicht die Leistung, aber das Lernpensum des Programmierers gesteigert wird.

Kapitel 11

Der 68000 im Detail

Interna

Supervisor-Modus

Exception-Processing

In diesem Kapitel soll etwas Hintergrundwissen zum 68000 vermittelt werden. Das brauchen Sie zwar nicht unbedingt zum Programmieren, aber sicherlich ist Ihnen auch wohler, wenn Sie wissen, warum Sie was tun.

Wahrscheinlich haben Sie im Laufe des Buchs schon gemerkt, daß ich ein 68000-Verehrer bin. Das ist nun nicht so aus der Luft gegriffen, sondern basiert auf langjähriger Erfahrung in der Programmierung anderer CPU's, nach dem Motto: Wer das Schlechte kennt, weiß das Gute zu schätzen.

Der 68000 ist der erste Mikroprozessor, dessen Befehlssatz an den der Mini-Computer angelehnt ist. Wenn man bedenkt, daß so moderne Rechner wie der ST mit seinem Hauptspeicher von 1,2 Megabyte (davon 200 K im ROM) den Minis der 70er Jahre inzwischen überlegen ist, ein 68020-System inzwischen schon eine VAX in Teilbereichen schlägt, dann ist dieser Befehlssatz auch die einzige Alternative. Leistungsfähige Rechner brauchen nämlich sehr komplexe System-Software, die nur sicher und effektiv zu erstellen ist, wenn die CPU die entsprechenden Grundlagen bietet.

11.1 Die innere Struktur des 68000

Prinzipiell besteht eine CPU immer aus einem Steuerwerk und einem Rechenwerk, man sagt auch »Control Logic« und »Arithmetic Logic Unit« (ALU). Das Steuerwerk besteht aus

Befehls-Register
und Befehls-Decoder.

Der Befehls-Decoder gibt seine Ergebnisse an die Ausführungseinheit, wo er zum Beispiel ALU-Funktionen oder Registerauswahl anstößt. Kern der Sache ist nun die Befehlsdekodierung. Die ersten Mikroprozessoren, wie zum Beispiel der legendäre F8, waren im Prinzip nichts weiter als programmierbare Logik-Bausteine. Bestimmte Bit-Muster an den Eingängen erzeugen da andere Bitmuster an den Ausgängen. In diesem Sinne waren Befehle auch nur Bitmuster, die per Hardware (mit vielen Gattern) dekodiert wurden. Die Technik läßt sich bei einer 8-Bit-CPU vielleicht heute noch vertreten, führt aber bei einem so mächtigen Befehlssatz wie dem des 68000 schnell in eine Sackgasse. Die Hardware wird dann nämlich bald unüberschaubar, verbraucht sehr viel Platz und ist kaum noch zu ändern. Der Ausweg ist der sogenannte

Mikro-Code.

Vereinfacht ausgedrückt heißt das, daß die Befehle des 68000, wie wir sie kennen, aus Sicht der CPU schon eine Hochsprache sind. Die CPU übersetzt mittels eines Programms diese von außen kommenden Makro-Befehle in eine Folge von internen Mikro-Befehlen. Das Steuerwerk der CPU ist demnach nicht als Hardware-Logik, sondern als Programm ausgelegt. Dieses Programm befindet sich in einem ROM-Bereich auf dem CPU-Chip. Da aber letztendlich doch Hardware in der CPU anzusprechen ist, ergibt sich ein Problem. Es sind bei jedem Befehl sehr viele »Bits« zu setzen, zum Beispiel schon 64, wenn zwei Register angesprochen werden. Macht man nun den Mikro-Code sehr schmal (zum Beipiel 4 Bit breit), benötigt man sehr viele Mikro-Zyklen, also viel Zeit, zur Auflösung eines Makrobefehls. Man spricht hier vom vertikalen Mikro-Code. Macht man den Mikro-Code

horizontal breiter, entstehen weniger Zyklen. Jetzt ist aber der Dekoder (zeit)aufwendiger. Man kann nun zwischen beiden Arten (vertikal oder horizontal) einen Kompromiß finden oder – noch besser – beide miteinander kombinieren.

Genau das macht der 68000. Es gibt einen Mikro-Code-ROM (Befehlsbreite 9 Bit) und einen Nano-Code-ROM (70 Bit). Die Mikro-Codes sind im Prinzip nur Zeiger auf die Nano-Codes, womit sich eine sehr schnelle Dekodierung ergibt, ähnlich, als wenn man in einem Buch nicht alle Seiten durchblättert, sondern zuerst im Inhaltsverzeichnis nachsieht.

Trotzdem ist die reine Hardware-Logik schneller, weil es dort prinzipiell keine Suchzeiten gibt. Um diesen Nachteil auszugleichen, hat man sich etwas einfallen lassen, was

Prefetch

heißt. Prefetch bedeutet soviel wie »vorab holen«. Ein Befehl kann ja aus bis zu fünf Worten bestehen. Der allgemeine Zyklus lautet

Holen
Dekodieren
Ausführen.

Überlappt man diese Vorgänge, spart man natürlich Zeit. Praktisch holt der 68000 bei der Abarbeitung eines Befehls schon das nächste Befehlswort und das darauffolgende. Dann wird immer ein neues Wort eingelesen, wenn/während eins abgearbeitet wird.

11.2 User- und Supervisor-Modus

Wie schon geschildert, gibt es beim 68000 den User- und den Supervisor-Modus. Der Vorteil dieser Trennung ist klar. Kernroutinen des Betriebssystems können nicht durch einen Fehler in einem User-Programm gestört werden. Nur dieses ermöglicht uns »Usern«, überhaupt solchen Fehlern auf die Spur zu kommen. Denn wie soll zum Beispiel ein Debugger uns die Registerinhalte anzeigen, wenn die dafür erforderlichen Betriebssystem-Routinen durch das fehlerhafte Programm zerstört wurden?

In welchem Modus sich der 68000 befindet, entscheidet nur ein Bit, nämlich das S-Bit im Statusregister. Wenn der ST durch einen Kaltstart (Einschalten) oder Warmstart (Reset-Taste) anläuft, befindet er sich automatisch im Supervisor-Modus. Der Übergang vom Supervisor- in den User-Modus kann erfolgen durch

RTE
Änderung des Supervisor-Bits
(MOVE #K,SR / ANDI #K,SR u.a.)

Vom User-Modus in den Supervisor-Modus gelangt man durch

Interrupt
Trap-Befehl
Exception (z. B. Adreß-Error)

Im User-Modus stehen alle 8 Datenregister, die 7 Adreßregister A0 bis A6, der Stackpointer (USP) und der PC zur Verfügung. Keinerlei Zugriff besteht auf den Supervisor-

Stackpointer (SSP). Eingeschränkt ist der Zugriff auf das Statusregister. Dieses besteht aus dem Supervisor-Byte, auch System-Byte genannt, und dem User-Byte (CCR = Condition Code Register). Auf das CCR besteht voller Zugriff, auf das System-Byte kann im User-Modus nur lesend zugegriffen werden.

Trace-Bit

Einen besonderen Komfort bietet das Trace-Bit. Ist dieses Bit (Bit 15 im Statusregister) gesetzt, geht der 68000 nach jedem Befehl in eine Exception, springt also zur Adresse, die im Trace-Vektor (Vektor 9 auf Adresse $24) eingetragen ist. Damit ist eine Einzelschrittbearbeitung möglich. Die Routine, auf die der Trace-Vektor zeigt, kann dann zum Beispiel die Register-Inhalte anzeigen. Anders ausgedrückt: Der schwierigste Teil eines Debuggers ist beim 68000 schon eingebaut. Sie können sich sicherlich vorstellen, daß Tracing ohne dieses Feature recht aufwendig zu programmieren ist. Man muß dann nämlich im zu testenden Code immer auf den nächten Befehl einen Sprung zur Trace-Routine legen, dann diesen Befehl wieder durch das Original ersetzen, den Trace-Sprung verlegen usw. Dazu muß man natürlich wissen, wieviel Bytes jeder Befehl belegt, sprich, einen Disassembler mitlaufen lassen.

11.3 Exceptions

Wir haben nun Exceptions schon so oft angewandt (mit jedem Trap #1), daß wir uns dieses wichtige Feature einmal genauer ansehen sollten.

Eine Exception ist eine Ausnahme, wenn man einmal nur das Wort als solches übersetzt. In den Ausnahmezustand kann der 68000 auf drei Arten gebracht werden:
1. durch externe Signale (Interrupt, Busfehler, Reset),
2. durch Fehler (zum Beispiel Adreßfehler),
3. »mit Absicht«.

Der dritte Fall ist der einfachste. Wir wenden ihn laufend mit dem Trap-Befehl an. Reset oder Interrupt sind auch noch einfach zu erklären, dazu muß man halt nur die entsprechenden Eingänge des 68000 ansteuern.

Bus-Error

Der Busfehler ist schon etwas komplizierter.

Beim Atari ST gibt es eine Baugruppe namens Memory Management Unit (MMU), die zuerst dafür sorgt, daß die Adressen, die der 68000 generiert, auch die richtigen Speicherchips ansprechen. So eine MMU hat immer einen Fehler(Fault)-Ausgang, der dann aktiviert wird, wenn eine nicht existierende Adresse angesprochen wird (der ST hat »nur« eines von den 16 möglichen Megabytes) oder aber ein geschützter Bereich. Der Fault-Ausgang der MMU ist mit dem Bus-Error-Eingang des ST verbunden.

Für uns besonders interessant ist nun die Tatsache, daß die ersten zwei KByte des ST-Speicherbereichs derart geschützt sind, daß ein Bus-Error erzeugt wird, wenn das Programm im User-Modus ist und versucht wird, auf eine Adresse in diesem Bereich zuzugreifen. Im Kapitel 12 wird geschildert, wie man dennoch an diesen Bereich herankommt, denn das

muß man, wenn man auf die dort abgelegten System-Variablen (siehe Anhang 8) zugreifen will.

Was passiert bei einer Exception?

Die Frage kann man à la Radio Erivan beantworten: Es kommt darauf an. . . Tatsächlich gibt es zwei Klassen von Exceptions. Allgemein wird immer der Programmzähler (PC) und das Statusregister (SR) auf dem Stack abgelegt. Bei einem Bus- oder Adreß-Error kommen noch drei Informationen hinzu, nämlich

- o der Code des gerade abgearbeiteten Befehls,
- o die Adresse, auf die gerade zugegriffen wurde,
- o das Super-Statuswort.

Im Super-Statuswort sind die Bits 0 bis 4 relevant und zwar

Bit 0..2: Funktions-Code
Bit 3: 0=Gruppe 2, 1=Gruppe 1
Bit 4: 0=Schreibzyklus wurde unterbrochen
1=Lesezyklus

Nun wäre wieder einiges zu erklären. Die Funktions-Codes sind 3 Ausgänge des 68000, mit der die CPU der MMU anzeigt, was sie gerade tut. Im wesentlichen erfolgt hier die Unterscheidung in Zugriffe auf Anwender-Daten, Anwender-Programm, Supervisor-Daten und Supervisor-Programm. Dahinter steckt, daß die CPU natürlich weiß, ob sie im Supervisor- oder User-Modus ist oder ob der PC auf ein Befehlswort oder ein Datum zeigt.

Die Gruppen 0, 1 und 2 haben folgenden Sinn. Überlegen Sie einmal, was passiert, wenn der 68000 in einer Exception-Bearbeitung ist und gerade dann noch eine Exception auftritt!

Nun, für so etwas gibt es Prioritäten. Die höchste Priorität hat die Gruppe 0, dann folgen 1 und 2. Gruppiert wird so:

Gruppe 0 Reset
Bus-Error
Adreß-Error

Gruppe 1 Trace
Interrupt
Illegaler Befehl
Trap $Axxx, Trap $Fxxx
Privilegverletzung

Gruppe 2 Trap #n
Trapv
CHK
Division durch 0

Man kann die Liste auch in einem Stück sehen. Danach hat dann Reset die höchste und »Division durch Null« die niedrigste Priorität. Tritt nun zum Beispiel während einer Exception eine solche mit höherer Priorität auf, wird das laufende Programm unterbrochen, die Routine mit der höheren Priorität abgearbeitet und dann die unterbrochene Routine fortgesetzt.

Exceptions beim Atari ST

Wie schon geschildert, wird nach einer Exception einiges an Informationen auf dem Stack gesichert und dann der PC mit der Adresse geladen, die im zugehörigen Vektor steht. Das wirkt dann wie ein Sprung zu dieser Adresse. Dehalb sollte in jedem Vektor schon etwas eingetragen sein, damit der 68000 nicht »in den Wald läuft«. Im Vektor 0 steht, welchen Wert der Stackpointer nach Reset einnehmen soll, im Vektor 1 der dann erste Stand des PC's. Diese Vektoren sind obligatorisch und müssen im Einschalt- oder Reset-Augenblick vorhanden sein. Da beim Einschalten ein RAM natürlich leer ist, muß per Hardware dafür gesorgt werden, daß dann »etwas ROM« auf diesen Adressen liegt. Die übrigen Vektoren kann nun jeder Systemprogrammierer nach Gusto belegen. Alle wird er kaum benötigen. Die dann freien Vektoren sollten nun aber zumindest alle mit einer (ein und derselben) Adresse geladen werden. Die Routine auf dieser Adresse kann schlicht mit RTE beginnen (und enden) oder eine kleine Meldung der Art »Vektor Nummer X nicht belegt« ausgeben. Anstatt dieser Meldung zeigt nun der ST gleich, daß er eine Grafik-Maschine ist und malt Bomben auf den Bildschirm. Dabei entspricht die Anzahl der Bomben der Vektor-Nummer. Zwei oder drei Bomben (Bus- oder Adreß-Fehler) erzeugt man wohl des öfteren.

Beim ST ist es dank der BIOS-Funktion Nummer 5 recht einfach, einen Vektor zu verbiegen, wie Bild 11.1 zeigt.

```
* VEK    Vektor Trap 7 setzen

        pea     neu          ; Adresse der neuen Routine
        move    #39,-(sp)    ; Vektor 39 = Trap #7
        move    #5,-(sp)     ; BIOS-Funktion SETEXEC
        trap    #13          ; aufrufen
        addq.l  #8,sp

        trap    #7           ; den neuen Trap aufrufen

        move    #1,-(sp)     ; warte auf Taste
        trap    #1
        addq.l  #2,sp
        clr     -(sp)        ; und Return zum Desk
        trap    #1
neu     pea     msg          ; gebe Text aus
        move    #9,-(sp)
        trap    #1
        addq.l  #6,sp
        rte

        data
msg     dc.b    'Hier ist Trap #7 ',0

        end
```

Bild 11.1 Verbiegen eines Vektors

Die neue Routine soll nur den Text »hier ist Trap #7« ausgeben. Das tut sie auf altbewährte Weise mittels der GEMDOS-Funktion Nummer 9. Beachten Sie das RTE am Schluß, das sollte sein, sonst bleibt ein Wort (Inhalt von SR) auf dem Stack. Um den Vektor für Trap 7 zu verbiegen, muß man nur der BIOS-Funktion 5 die Adresse der neuen Routine verraten, noch die Vektor-Nummer nennen und »BIOS 5« aufrufen. So einfach ist das. Vielleicht schreiben Sie einmal auf dieser Basis ein kleines Programm, das bei den Fehlern Bus-Error (Vektor 2) und Adreß-Error (Vektor 3) eine passende Meldung ausgibt und auf eine Taste wartet. Der ST kehrt nämlich in der Regel nach der »Bombenanzeige« so schnell zum Schreibtisch zurück, daß man kaum eine Chance hat, die Bomben zu zählen. Bei drei geht's ja noch, aber Sie können das Programm ja weiter ausbauen.

Die komplette Vektor-Liste finden Sie im Anhang 9.

Kapitel 12

Utility 1

Ein RAM-Disk-Programm

Die vollautomatische RAM-Disk

12.1 High Speed: Ein RAM-Disk-Programm

Wenn man ein Megabyte RAM hat, fällt es schon schwer, dafür sinnvolle Anwendungen zu finden. Hier ist eine: 100 K werden in einer Sekunde geladen, Compilerzeiten fallen kaum noch auf, und Disketten werden blitzschnell kopiert.

Das Geheimnis heißt RAM-Disk. Für den Benutzer sieht eine RAM-Disk wie eine normale Diskette aus. Sie erscheint auf dem Schreibtisch mit dem üblichen Abbild einer Diskstation, zum Beispiel als Drive C. Doch es rotiert keine Scheibe mehr, kein Schreib/Lesekopf sucht noch die richtige Spur und Geräusche macht sie auch nicht. Praktisch wird die ganze Diskette im RAM simuliert. Sind die Daten erst einmal auf der RAM-Disk, heißt Lesen oder Schreiben nur noch Bewegen der Daten im RAM, und das kann die 68000-CPU bekanntlich blitzschnell. Aber auch, wenn Sie Daten von einer Diskette auf die RAM-Disk schreiben (oder umgekehrt), geht das immer noch mehr als doppelt so schnell wie der Transfer zwischen 2 Disketten.

Einen Nachteil hat die Geschichte. Wenn Sie den Rechner ausschalten, sind alle Daten auf der RAM-Disk gelöscht. Also bitte vorher auf eine langsame (aber echte) Diskette kopieren! Um gleich bei der Bedienung zu bleiben: Sie brauchen die RAM-Disk nicht zu formatieren, genau: Sie dürfen das gar nicht! Eine Disk-Copy (eine Diskette auf die andere schieben) funktioniert auch nicht, und ist auch nicht sinnvoll. Stattdessen wählen Sie alle Files auf der Quell-Diskette aus. Dazu können Sie mit der Maus bei gedrückter Taste das sich dann bildende Rechteck über die Auswahl ziehen oder bei gedrückter Shift-Taste einzelne Files auswählen. Danach schieben Sie eines der Files auf die RAM-Disk, die anderen folgen von selbst.

Genauso können Sie RAM-Disk-Dateien löschen (Auswahl in den Papierkorb) oder die Dateien auf der RAM-Disk sichern (Auswahl auf eine echte Disk). Bliebe noch zu klären, wie man die RAM-Disk installiert. Ganz einfach: Sie klicken das Programm RAMDISK.PRG an und folgen der dann erscheinenden Bedienungsanleitung. Nach dem »Anmelden« sollten Sie auf dem Bildschirm das Abbild von RAMDISK sehen. Wenn nicht, wird es von einem Fenster verdeckt. Also schließen Sie die anderen Diskettenfenster (oder tun Sie das schon vorher).

Nun können Sie die RAM-Disk anklicken und auf eine beliebige Stelle des Schreibtisches schieben. Wenn Sie nun (RAM-Disk ist noch schwarz) Info aus dem Menü ziehen, sollte die von Ihnen gewählte Größe (wir kommen noch drauf) ablesbar sein. Noch eine Warnung: Wenn Ihnen die Größe nicht gefällt, sollten Sie das RAM-Disk-Programm nicht einfach wieder starten. Es wird dann nämlich noch eine RAM-Disk installiert und irgendwann ist auch ein Megabyte-Speicher erschöpft.

Wie es funktioniert?

Unter den Systemvariablen (siehe Anhang 8) gibt es einige Vektoren, die mit hdv_ anfangen, was für »hard disk vector« steht. In diese Links wird normalerweise eine Hard-Disk eingebunden, genau hier haken wir unsere RAM-Disk ein. Von den 5 Vektoren brauchen wir 3 (booten von der RAM-Disk geht leider nicht), nämlich hdv_bpb, womit die Adresse des BIOS-Parameter- Blocks ermittelt wird, ferner hdv_rw, worüber das read/write von Sektoren läuft und letztlich noch hdv_mediachange. Ich melde hier sicherheitshalber,

daß das Medium (die RAM-Disk) nicht gewechselt wurde. Das Programm lt. Bild 12.1 funktioniert im Prinzip so:

Wenn das Programm geladen wird, stehen damit auch schon die neuen Routinen im RAM. Das Programm läuft dann los und tut folgendes:
1. Nach der Größe der RAM-Disk fragen,
2. RAM-Disk einrichten,
3. Speicherbedarf ermitteln,
4. die hdv_-Vektoren auf die neuen Routinen verbiegen,
5. Anwender-Info ausgeben,
6. zurück zum Schreibtisch, dabei aber den Speicher nicht freigeben.

Das Programm wurde einst für Computer persönlich geschrieben, deshalb macht es mit einem entsprechenden Logo auf. Danach wird die Größe der RAM-Disk in 100-K-Schritten erfragt. Dadurch ist nur eine einstellige Eingabe erforderlich, die die übliche Umwandlung von Strings in Zahlen erspart, genau: drastisch verkürzt. Trotzdem wird zum Beispiel die Eingabe von »5« mit »500« hinterfragt, indem ganz einfach die Zeichen »00« mit ausgegeben werden.

Ab »Start« wird der Anwender nach der Größe gefragt. Hat er sich entschieden, ist aber nicht in den erlaubten Grenzen geblieben, erfolgt ein Rücksprung zu »redo«. Beachten Sie, daß gleich nach dem Label »start« ein »bra ask« folgt. Damit wird der »Beep« (Warnton) übersprungen, der als einzige Reaktion auf eine falsche Eingabe erzeugt wird.

Nach diesem Test folgt die Rückfrage. Da hier nur die Antworten »J« (Ja) und »A« (Abbruch) ausgewertet werden, ist dieser Job recht einfach. Ab Label »OK« wird die Größe, die ja bis jetzt nur ein Zeichen im Bereich '1' bis '8' ist, in eine Zahl (100 bis 800) gewandelt.

Ab »RAM-Disk einrichten« nähern wir uns dem Kern des Programms. Sie wissen noch aus Kapitel 8, daß auf den ersten beiden Tracks einer Diskette (den ersten 18 Sektoren) der Bootsektor, die FATs und das Directory stehen. Das meiste davon ist bei einer neuen Diskette leer, weshalb hier dieser Bereich zuerst einmal global mit Nullen gefüllt wird. Das geschieht in einer DBcc-Schleife mit D0 als Zähler. Der Ausdruck

```
move   #18*512/4-1,d0
```

rührt her aus 18 Sektoren zu 512 Bytes. Dividiert durch 4 wird, weil mit Langworten gefüllt wird. Dieses nur, damit es (unmerklich) schneller geht.

Ab »Boot-Sektor schreiben« passiert folgendes:

Von der RAM-Disk kann man natürlich nicht »booten«, trotzdem braucht auch die RAM-Disk einen Boot-Sektor. Weil aber das TOS von einer Daten-Disk nur den BPB-Teil liest (siehe Kapitel 8.6), müssen wir auch diesen generieren. Das geschieht ganz einfach durch Kopieren einer Tabelle mit den Standard-Daten einer einseitigen Diskette. Zum Schluß müssen wir nur noch die aktuelle Größe unserer RAM-Disk eintragen. Hierzu eine Anmerkung: Die meisten Programme, die auf eine Diskette schreiben, testen vorher deren noch freien Platz und geben ggf. eine Fehlermeldung aus. Das funktioniert auch mit der RAM-Disk. Kommt die Meldung erst, wenn die Disk voll ist (ein Sektor paßt nicht mehr 'rauf), dann ist das ein Fehler des jeweiligen Programms.

Wie auch immer, wir haben jetzt die Größe der RAM-Disk und machen damit gleich weiter. Der Trick ist nämlich, daß die RAM-Disk deshalb permanent aktiv bleibt, weil wir nicht über GEMDOS Nummer 0 zurückkehren werden, sondern über die Funktion »keep«. Dieser übergibt man die Größe des Speichers, der reserviert werden soll. Danach kehrt auch »keep« zum Desktop zurück, doch der Speicherbereich ist nun für alle folgenden Programme gesperrt. In diesem Speicherbereich befindet sich das RAM-Disk-Programm und natürlich die RAM-Disk selbst. Wie groß das Programm mit seinen Daten-Segmenten ist, steht in der Base-Page (siehe Kapitel 8).

Beachten Sie, daß man bei der Addition aller Werte nicht die Base-Page selbst mit ihren 256 Bytes vergessen darf.

Nun müssen wir (vor der Rückkehr zum Desktop) nur noch die RAM-Disk dem TOS bekanntmachen. Das Prinzip ist folgendes:

Das gesamte Disk-I/O läuft über die Hard-Disk-Vektoren. Wenn Sie keine Hard-Disk haben, zeigen die Vektoren auf die normalen Disketten-Routinen im BIOS. Haben Sie eine Hard-Disk, zeigen die Vektoren auf die HD-Treiber, die die normalen Disketten-Befehle in HD-Befehle übersetzen. Die HD-Treiber selbst springen nach diesem Zwischenschritt wieder zum BIOS. In diese HD-Vektoren klinken wir uns nun ein, und zwar so:

Die ganze Routine ist ein Unterprogramm, das mittels der XBIOS-Funktion »super« aufgerufen wird. Das ist notwendig, weil wir auf Systemvariable zugreifen, die nur im Supervisor-Modus zugänglich sind. »super« wird nur die Adresse der Routine (hier Label patch) übergeben, in »patch« selbst geschieht folgendes:

Zuerst wird die im Vektor eingetragene Adresse gerettet. Dann wird in den Vektor die Adresse unserer Routine eingetragen. So etwas nennt man »Vektor verbiegen« oder »patch« (wörtlich: flicken). Unsere »eingeflickten« Routinen machen nun folgendes:

Sie prüfen, ob die RAM-Disk gemeint ist. Wenn nicht, springen sie zur alten Adresse (die, die vorher im Vektor stand), ansonsten »spielen« sie RAM-Disk. Das Prüfen ist recht einfach. Weil alle Routinen, die bei den Vektoren landen, BIOS-Funktionen aufrufen, und dazu auch immer die Drive-Nummer auf den Stack packen, müssen wir nur diese Drive-Nummer vom Stack lesen und prüfen, ob es die Nummer des RAM-Disk-Drives ist.

Damit wären die ersten 6 Zeilen ab Label Patch geklärt, nun kommt die Sache mit der »Drive Map«. In der Systemvariablen »drvbits« ist für jeden angemeldeten Drive ein Bit gesetzt, und zwar Bit 0 für Drive A, Bit 1 für Drive B u.s.w. Einfache RAM-Disk-Programme setzen nun einfach Bit 2 für Drive C und die Sache hat sich. Das funktioniert nun aber nicht mit einer Hard-Disk, denn die ist zuerst immer Drive C. Nun einfach auf D zu gehen klappt auch nicht, denn die Hard-Disks werden immer in mehrere logische Drives eingeteilt. Wir wissen also nicht, ob zum Beispiel D oder E der letzte Drive im System ist. Deshalb ist es sinnvoll, den letzten Drive zu suchen und dann die RAM-Disk darüberzulegen. Außerdem beinhaltet diese Methode die Möglichkeit, auch mehrere RAM-Disks einzurichten.

Im Prinzip ist die Sache wieder recht einfach. In einer kleinen Schleife wird mittels des BTST-Befehls getestet, welches Bit (von unten gesehen) noch nicht gesetzt ist. Danach ist der Stand des in der Schleife hochgezählten Zählers die Drive-Nummer und die Drive-Nummer plus 'A' dann der Drive-Name. Beides wird abgespeichert, da später noch benötigt.

Danach schiebe ich alle Bits in »drvbits« um eines nach links und setze Bit 0 wieder auf 1. Damit hätten wir den neuen Drive eingetragen.

Sie werden nun einwenden, daß dieses mit dem BSET-Befehl, der ja auch testet, aber dann das Bit gleich setzt, viel einfacher zu realisieren ist. Recht haben Sie, nur meine Methode hat den Vorteil, daß hier noch die Chance einer Rückfrage besteht (Wollen Sie wirklich Drive G?). Der Hintergedanke: In einem System mit ROM-TOS und Hard-Disk löscht ein Reset zwar die RAM-Disk, nicht aber das Drive-Bit!

Nun können wir uns den neuen Routinen zuwenden. Jede der drei Routinen hat grundsätzlich diesen Aufbau:

```
new_xx  move    dd(sp),d4     ; Drive-Nummer holen
        cmp     drive,d4      ; RAM-Disk?
        bne     alt_xxx       ; wenn nicht

*       neue Routine hier
        rts

notC1   move.l  old_bpb,a0    ; alte Adresse holen
        jmp     (a0)          ; und dahin
```

Beachten Sie bitte, daß ich mit »bne« auf »ungleich« teste, also frage, ob es nicht die RAM-Disk ist. Würden Sie die Logik umdrehen (mit beq testen und die beiden Abschnitte vertauschen) funktioniert die Sache sicherlich auch. Sie hätten dann nur gegen eine alte Programmiererregel verstoßen, die da lautet, »ein Test führt direkt zum Normalfall«.

Falls ein »Caller« den BIOS-Parameter-Block der RAM-Disk sehen will, ist die neue Routine nur eine Zeile, nämlich

```
        move.l  #bpbtab,d0
```

Damit wird ein Zeiger auf den BPB der RAM-Disk zurückgegeben. Noch einfacher ist die Sache im Falle von »media changed«, womit getestet wird, ob eine Diskette gewechselt wurde. Diese Frage können wir bei einer RAM-Disk getrost mit Nein beantworten, was bei dieser Funktion 0 heißt.

Daher reicht ein schlichtes »clr d0«, und schon ist das Thema erledigt. Bleibt die eigentliche RAM-Disk-Funktion, sprich, das Schreiben in den RAM anstatt auf eine Diskette und das Lesen aus dem RAM anstatt ...

Schauen wir uns zuerst einmal an, was ein Programm tut, das Daten auf eine Diskette schreiben oder von ihr lesen will. Es ruft immer direkt oder im Falle von GEMDOS-Routinen über einen Umweg, die BIOS-Funktion Nummer vier (rwabs) auf, und zwar so:

```
        move    #drvno,-(sp)   ; Drive: 0=A, 1=B u.s.w.
        move    #secno,-(sp)   ; relative Sektor-Nummer
        move    #seccnt,-(sp)  ; Anzahl Sektoren
        pea     bufadr         ; Pufferadresse
        move    #rwflag,-(sp)  ; 0=Lesen, 1=Schreiben
        move    #4,-(sp)       ; Funktions-Nummer
        trap    #13            ; BIOS aufrufen
```

In diesem Augenblick befinden sich folgende Daten auf dem Stack, wobei die Werte der Equates das Offset zum Stackpointer angeben. Glauben Sie mir bitte erst einmal, daß sich der letzte Parameter bei »4(sp)« findet. In Kapitel 15 zeige ich den Trap-Dispatcher, Sie werden dann den Grund verstehen.

```
rwflag    equ    4        ; Read/Write-Flag
bufadr    equ    6        ; Pufferadresse
seccnt    equ    10       ; Anzahl Sektoren
secno     equ    12       ; Sektor-Nummer
drvno     equ    14       ; Drive-Nummer
```

Mit diesen Informationen ausgestattet, können wir nun die erste Teilaufgabe lösen. »BIOS 4« meldet uns ja eine relative Sektor-Nummer. Die RAM-Disk ist aber nicht in Sektoren eingeteilt. Stattdessen sagen wir, Sektor 0 beginnt beim Label »ramdisk«, Sektor 1 bei »ramdisk + 512« u.s.w. Daraus ergibt sich die Startadresse auf der RAM-Disk nach der Formel

```
Start = Sektor_nummer * 512 + ramdisk
```

In Assembler sieht das dann so aus:

```
move      secno(sp),d0   ; Sektor-Nr. lesen
ext.l     d0             ; auf Long erweitern
move      #9,d1          ; 9 Bit Shift (*512) vorbe-
                           reiten
lsl.l     d1,d0          ; *512 = Offset in RAM-Disk
lea       ramdisk,a1     ; Start RAM-Disk
add.l     d0,a1          ; + Offset
                         ; = Adr. in RAM-Disk
```

Vorab hatte ich noch mit

```
move.l    bufadr(sp),a0
```

die Adresse des Puffers vom Stack geholt. A0 zeigt also auf den Puffer, A1 (siehe oben) auf den Beginn der RAM-Disk. Sollen Daten vom Puffer auf die RAM-Disk geschrieben werden, müßte jetzt so etwas wie ein

```
move      (a0)+,(a1)+
```

folgen. Soll hingegen von der RAM-Disk in den Puffer geschrieben werden, müßte man

```
move      (a1)+,(a0)+
```

schreiben. Ob gelesen oder geschrieben werden soll, steht im »rwflag«. Das muß man auf jeden Fall testen, nur danach zwei verschiedene Routinen anzuspringen, bringt eine glatte Fünf im Logik-Kurs, denn da war doch was ... Richtig! Quelle und Ziel müssen miteinander vertauscht werden, also:

```
        move     rwflag(sp),d0 ; Lesen oder Schreiben?
        btst     #0,d0         ; Lesen?
        beq      rdwrt         ; wenn so
        exg      a0,a1         ; sonst Q/Z-Tausch
rdwrt   ; hier geht's weiter
```

Der Rest ist nun recht einfach. Es müssen soviele Sektoren übertragen werden, wie in »seccnt« steht. Ein Sektor sind aber hier 512 Bytes. Wer die »512« wörtlich nimmt, hat Pech gehabt, er verschenkt nämlich viel Zeit. Wir sind jetzt nämlich im Kern der RAM-Disk und hier (und nur hier!) kommt es auf das Tempo an, schließlich ist die Geschwindigkeit der Hauptvorteil der RAM-Disk. Der Befehl »move.b (a1)+,(a0)+« benötigt 12 Clockzyklen, »move.l (a1)+,(a0)+« hingegen 20, aber:

512 * 12 ist 6144, 128 * 20 ist 2560. Demnach ist der Transfer mit Langworten 6144/2560 = 2.4 oder um 240 % schneller.

Schauen wir uns nun die Kernroutine an:

```
rdwrt   move    seccnt(sp),d1 ; Sector Count
        subq    #1,d1         ; -1 wegen dbr
rwloop  move    #127,d0       ; 128 Longs=1 Sektor
rw1     move.l  (a1)+,(a0)+   ; kopieren
        dbra    d0,rw1
        dbra    d1,rwloop     ; FOR Sector_Count
        moveq   #0,d0
```

Es laufen zwei DBcc-Schleifen, die äußere über die Anzahl Sektoren, die innere über die 128 Langworte eines Sektors. Zum Schluß steht da noch ein einsames »moveq #0,d0«. Damit melden wir dem aufrufenden Programm, daß kein Fehler aufgetreten ist. Das ist zulässig, denn es geht hier nur um den Transfer RAM nach RAM, und ich unterstelle schlicht, daß der RAM in Ordnung ist. Hier nun das vollständige Listing:

```
* RAMDISK Ram-Disk-Programm fuer Atari ST

gemdos  equ     1             ; gemdos trap
keep    equ     $31           ; ret & keep mem

xbios   equ     14            ; xbios trap
super   equ     38            ; exec in supervr-mode

* hdv_ = hard disk vektor:
hdv_bpb equ     $472          ; get bios parameter block
hdv_rw  equ     $476          ; read/write abs
hdv_mediach equ $47e          ; media changed
drvbits equ     $4c2          ; bit map active drvnames
*-------------------------------------------------------------
* Die folgenden Makros werden nur fuer den Dialog benoetigt

writes  macro                 ; write string
        pea     \1
        move.w  #9,-(sp)
        trap    #1
        addq.l  #6,sp
        endm
```

```
conin   macro                   ; get char -> d0
        move.w  #7,-(sp)
        trap    #1
        addq.l  #2,sp
        endm

conout  macro
        move.w  \1,-(sp)        ; Zeichen
        move.w  #2,-(sp)        ; Funktion CONOUT
        trap    #1              ; aufrufen
        addq.l  #4,sp           ; Stack korrigieren
        endm
*-------------------------------------------------------------
********** Hier geht es los ************

start   writes  logo            ; Logo und Query ausgeben
        bra     ask
redo    writes  beep            ; hierher falls Fehler
ask     conin
        cmpi    #'1',d0         ; teste '1'
        bmi     redo            ; wenn kleiner
        cmpi    #'8',d0         ; teste '8'
        bgt     redo            ; wenn groesser

        move    d0,d3           ; Groesse der RAMDISK in 100
                                  K
                                ; Hier Konstante nach d3,
                                  wenn kein Dialog, sonst
                                  hier weiter:
        move    d3,-(sp)        ; Antwort ausgeben
        move    #2,-(sp)        ; d. h. erste Ziffer
        trap    #1
        addq.l  #4,sp

        writes  prompt          ; und '00' + Frage ob OK
        conin                   ; Antwort einlesen
        bclr    #5,d0           ; force uppercase
        cmpi    #'A',d0         ; Abbruch
        bne     weiter          ; wenn nicht
        clr     -(sp)           ; sonst Return
        trap    #1              ; zum Desk

weiter  cmpi    #'J',d0         ; Wenn nun nicht J
        bne     start           ; noch einmal

ok      sub     #48,d3          ; '1'..'8' in 1..8
        ext.l   d3              ;
        mulu    #100,d3         ; und mal 100
```

```
* RAM-Disk einrichten
* ===================
* Tracks 0,1 loeschen
* -------------------
        lea     ramdisk,a0
        move    #18*512/4-1,d0 ; 2 Tracks in Langworten
Dloop   clr.l   (a0)+          ; mit Nullen laden
        dbra    d0,Dloop

* Boot-Sektor schreiben
* ---------------------
        lea     ramdisk+11,a0  ; 0..10 ist schon 0
        lea     bootsec,a1
        move    #seclen-1,d0
bloop move.b (a1)+,(a0)+       ; Tabelle kopieren
        dbra    d0,bloop
        move    d3,numcl       ; Groesse der Disk in K
        move    d3,d2          ; retten

* Anzahl Sektoren ermitteln und eintragen
* ----------------------------------------
        lsl     #1,d3          ; *2=Groesse in Sektoren
        add     #18,d3         ; +18 fuer das Management
        lea     ramdisk+19,a0  ; =Gesamt Sektoren
        move.b  d3,(a0)+       ; LSByte im 8086-
        lsl     #8,d3          ; Format
        move.b  d3,(a0)        ; MSByte eintragen

* Speicherbedarf ermitteln
* ------------------------
        moveq   #10,d0         ; *1024 vorbereiten
        lsl.l   d0,d2          ; Groesse in Bytes rechnen
        move.l  4(sp),a0       ;     Zeiger auf Base Page
        add.l   12(a0),d2      ; + Laenge Text
        add.l   20(a0),d2      ; + Laenge Data
        add.l   28(a0),d2      ; + Laenge BSS
        add.l   #$100,d2       ; + Laenge Base Page
                               ;= Gesamtbedarf

* in Supervisor-Mode gehen, da Zugriff auf System-Vars.
* -----------------------------------------------------
        pea     patch          ; Adr. Patch-Routine
        move    #super,-(sp)
        trap    #xbios
        addq.l  #6,sp

* Manual und Return -> Desktop, Speicher behalten
* -----------------------------------------------
        writes  manual         ; Bedienungsanleitung,
```

```
        conout  drvname         ; Drive-Buchstaben und
        writes  manual1         ; und noch etwas Text
        conin                   ; auf Taste warten
        clr     -(sp)
        move.l  d2,-(sp)        ; Speicherbedarf
        move    #keep,-(sp)     ; anmelden
        trap    #gemdos         ; und Return Desk / Caller

* Vektoren patchen und Drive eintragen
* ------------------------------------
patch   move.l  hdv_bpb,old_bpb     ; getbpb-Vektor retten
        move.l  #new_bpb,hdv_bpb    ; dann patchen

        move.l  hdv_rw,old_rw       ; dto. rw-Vektor
        move.l  #new_rw,hdv_rw

        move.l  hdv_mediach,old_med ; dto.media-change-Vektor
        move.l  #new_med,hdv_mediach

* Drive suchen/setzen
* -------------------
        move.l  drvbits,d0      ; get drive bits
        move    #0,d4           ; count=0
loop    add     #1,d4
        btst    d4,d0           ; drive bit set?
        bne     loop
        move    d4,drive        ; Drive-Nummer merken
        add     #'A',d4         ;  + 'A'
        move    d4,drvname      ;  = Drive-Buchstabe
; hier kann man noch testen, ob Drive OK.
; tun wir nicht, also:
        lsl.l   #1,d0           ; 1 Bit links mehr
        or.l    #1,d0           ; das rechte wieder auf 1
        move.l  d0,drvbits      ; nun aber!
        rts

* Die eigentlichen RAM-Disk-Routinen starten hier
* ===============================================
* get bpb: Wenn BIOS-Parameter Block angefordert wird
* -------
new_bpb move    4(sp),d4        ; Drive-Nummer holen
        cmp     drive,d4        ; RAM-Disk?
        bne     notC1           ; wenn nicht
        move.l  #bpbtab,d0      ; return RAM-Disk's bpp
        rts
notC1   move.l  old_bpb,a0      ; alte Adresse holen
        jmp     (a0)            ; und dahin

* read/write sector: Wenn Disk-I/O angefordert wird
* -----------------
```

```
; Dies ist die BIOS-Funktion 4 sozusagen von der anderen
; Seite her gesehen. Auf dem Stack sind bei xx(sp):

rwflag  equ     4               ; Read/Write-Flag
bufadr  equ     6               ; Pufferadresse
seccnt  equ     10              ; Anzahl Sektoren
secno   equ     12              ; Sektor-Nummer
drvno   equ     14              ; Drive-Nummer

new_rw  move    drvno(sp),d4
        cmp     drive,d4        ; RAM-Disk?
        bne     notC2           ; if not

        move.l  bufadr(sp),a0   ; Adr. Sektor-Puffer
        move    secno(sp),d0    ;  rel. Sektor-Nr.
        ext.l   d0              ; -> long
        move    #9,d1           ; *512
        lsl.l   d1,d0           ; =Offset in RAM-Disk
        lea     ramdisk,a1      ; Start RAM-Disk
        add.l   d0,a1           ; + Offset
                                ; = Adr. in RAM-Disk
        move    rwflag(sp),d0   ; Lesen oder Schreiben?
        btst    #0,d0           ; Lesen?
        beq     rdwrt           ; wenn so
        exg     a0,a1           ; nein:Ziel/Quelle tauschen

rdwrt   move    seccnt(sp),d1   ; Sector Count
        subq    #1,d1           ; -1 wegen dbra
rwloop  move    #127,d0         ; 128 Longs=1 Sektor
rwl     move.l  (a1)+,(a0)+     ; kopieren
        dbra    d0,rwl
        dbra    d1,rwloop       ; FOR Sector_Count
        moveq   #0,d0
        rts

notC2   move.l  old_rw,a0
        jmp     (a0)

* Media changed
* -------------
new_med move 4(sp),d4           ; Drive holen
        cmp     drive,d4        ; RAM-Disk?
        bne     notC3
        clr     d0              ; RAM-Disk kann natuerlich
        rts                     ; nicht gewechselt werden
notC3   move.l  old_med,a0
        jmp     (a0)

        data
logo    dc.b      27,'E',13,10  ; cls & crlf
```

```
        dc.b    27,'p'          ; reverse on
        dc.b    '                       Computer            '
        dc.b    27,'q',27,'Y',33,88,'Programm-Service'
        dc.b    13,10,27,'Y',34,88,'fuer ATARI 520 ST'
        dc.b    27,'p',13,10
        dc.b    '                           persoenlich     '
        dc.b    27,'q',13,10,13,10
        dc.b    13,10
        dc.b    '  R A M - D i s k fuer ATARI xxx ST',13,10
        dc.b    '  ================================',13,10
        dc.b    '  (c) 1985 Peter Wollschlaeger'
        dc.b    13,10,13,10
        dc.b     ' Auf einem 260 ST kann die RAM-Disk nur
                100 K Byte gross sein:
        dc.b    13,10
        dc.b    ' auf einem ST+ bis zu 600 K (mit ROM
                jeweils 200 K mehr).'
        dc.b    13,10,13,10,13,10
        dc.b    ' Waehlen Sie 1..8 fuer 100..800 K Byte: '
        dc.b    0

beep    dc.b    7,0
prompt  dc.b    '00 K Byte OK (J/N A)bruch) ? ',0
manual  dc.b    27,'Y',38,32,27,'J'
        dc.b    'Wenn Sie gleich zum Schreibtsch zurueck-
                kehren,'
        dc.b    13,10
        dc.b    'schliessen Sie alle Fenster und'
        dc.b    13,10
        dc.b    'klicken Sie Diskstation A an (so dass die
                schwarz wird).'
        dc.b    13,10
        dc.b    'Nun waehlen aus dem Menu Optionen Disk-
                station anmelden.'
        dc.b    13,10
        dc.b    'Dort waehlen Sie als Bildbezeichnung RAM-
                DISK'
        dc.b    13,10
        dc.b    'und Laufwerk ',0
manual1 dc.b    13,10,'Druecken Sie eine Taste',0
        ds.w    0
drive   ds.w    1
drvname ds.w    1

bpbtab  dc.w    $200            ; Standard BPB
        dc.w    2
        dc.w    $400
        dc.w    7
```

```
        dc.w    5
        dc.w    6
        dc.w    18
numcl   ds.w    1               ; hierher Groesse RAM-Disk
        ds.w    8

bootsec dc.b    0,2             ; Boot-Sektor ab Byte $0B
        dc.b    2
        dc.b    1,0
        dc.b    2
        dc.b    112,0
        ds.b    2
        dc.b    0
        dc.b    5,0
        dc.b    9,0
        dc.b    1,0
        dc.b    0
seclen  equ     *-bootsec

        bss
old_bpb ds.l    1
old_rw  ds.l    1
old_med ds.l    1
ramdisk equ     *

        end
```

Bild 12.1 Das RAM-Disk-Programm

12.2 Die vollautomatische RAM-Disk

Laut der eingangs geschilderten Bedienungsanleitung bleibt Ihnen immer noch die Aufgabe, die Frage nach der Größe zu beantworten und dann die RAM-Disk anzumelden. Zum zweiten werden Sie wohl nach jedem Kaltstart die RAM-Disk erst einmal »füttern« müssen, sprich, als Assembler-Programmierer, zumindest Editor, Assembler und Linker auf die RAM-Disk kopieren. Um das zu automatisieren, müssen Sie zuerst festlegen, welche Größe die RAM-Disk jeweils haben soll. 500 K wäre ein guter Wert.

So ziemlich am Anfang des Listings steht mit entsprechendem Kommentar »move d0,d3«. Das ersetzen Sie durch »move #5,d3« (oder Ihre Wahl) und streichen den ganzen Dialog davor und alle zugehörigen Texte am Schluß des Listings.

Nun müssen Sie sich entscheiden, welchen Drive Ihre RAM-Disk immer annehmen soll. In der Routine »Drive-Bit suchen/setzen« ersetzen Sie das

```
        move.l  d0,drvbits
```

durch ein schlichtes

```
        bset    #n,drvbits
```

Natürlich können Sie nun den Aufwand davor auch streichen. Für »n« gilt: Drive C=2, D=3 usw.

Lassen Sie jetzt das Programm einmal laufen, melden die RAM-Disk an, ordnen den Schreibtisch und geben »Arbeit sichern«. Nun kopieren Sie noch die RAM-Disk in den Auto-Ordner und beim nächsten Start ist die RAM-Disk da.

Nächster Schritt: Im Kapitel 8 wurde ein Kopier-Programm zur Auto-RAM-Disk vorgestellt. In diesem Programm müssen Sie an einer Stelle den Drive-Buchstaben Ihrer RAM-Disk eintragen, wenn es nicht schon »C« ist. Jetzt kopieren Sie dieses Programm auch in den Auto-Ordner. Da Programme in diesem Ordner in der Reihenfolge des Eintrags ausgeführt werden, müssen Sie unbedingt darauf achten, daß das Kopierprogramm nach dem RAM-Disk-Programm in den Autoordner gelangt.

Dritter Schritt: Richten Sie einen Ordner mit dem Namen RDFILE ein. In diesen kopieren Sie alle Programme, die in die RAM-Disk sollen.

Letzter Schritt: Wahrscheinlich wollen Sie Ihren Assembler auch gleich von der RAM-Disk starten, das RAM-Disk-Fenster sollte also offen sein. Dazu öffnen Sie das Fenster und ordnen es so auf dem Schreibtisch an, wie Sie es gerne hätten. Nun geben Sie »Arbeit sichern«. Daraufhin erscheint das Desktop-Info-File im RAM-Disk-Fenster. Kopieren Sie es von dort auf die Startdiskette. Das war's. Beim nächsten Start macht der Schreibtisch mit dem RAM-Disk-Fenster auf, alle Files sind dort schon sichtbar und Sie brauchen nur noch auf ASSEM.PRG zu klicken oder wie Ihr Lieblingsprogramm sonst heißen mag. Letzter Tip: Für verschiedene Anwendungen habe ich mir einfach verschiedene Startdisketten in dieser Art eingerichtet.

Kapitel 13

Utility 2

Ein Diskmonitor

Bildschirmgestaltung

Menütechnik

Displays

XBIOS

Es gibt einen wirklich sehr guten Diskeditor für den ST, das ist T.L.D.U., »The Last Disk Utility« von Focus Computer. Dieses Programm kann ich Ihnen empfehlen. Sie wollen aber wissen, wie solche Routinen im Kern funktionieren und Assembler lernen, deshalb soll hier ein Diskmonitor vorgestellt werden.

```
Drive=0    Side=0    Track=10    Sector=2                    Operation OK

0000 000A 7E6E 756D 0000 0000 C000 FFFF FFFC 7E70 6172 7469 6400 C000 FFFF FFF8
7E69 6473 7472 0000 C000 FFFF FFF4 5F67 7061 7274 0000 A200 0000 2B2E 7E7E 6770
6172 7400 8200 0000 2B2E 7E69 6D61 6765 0000 C000 0000 0003 7E70 696E 666F 0000
C000 0000 000C 7E72 7061 7274 0000 D000 0000 000D 7E69 0000 0000 0000 C000 FFFF
FFFE 7E6A 0000 0000 0000 C000 FFFF FFFC 5F73 7061 7274 0000 A200 0000 2BAE 7E7E
7370 6172 7400 8200 0000 2BAE 7E69 6D61 6765 0000 C000 0000 0003 7E70 696E 666F
0000 C000 0000 000C 7E72 7061 7274 0000 D000 0000 000D 7E69 0000 0000 0000 C000
FFFF FFFE 7E6A 0000 0000 0000 C000 FFFF FFFC 5F64 6F70 6172 7400 A200 0000 2C2E
7E7E 646F 7061 7274 8200 0000 2C2E 7E70 696E 666F 0000 D000 0000 000D 7E70 6879
7364 6576 C000 0000 0003 7E69 0000 0000 0000 C000 FFFF FFFE 7E69 6D61 6765 0000
C000 FFFF FDFE 7E66 6174 7369 7A00 C000 FFFF FDFA 7E6E 6469 7273 0000 C000 FFFF
FDF6 7E62 7300 0000 0000 C000 FFFF FDF2 5F69 7700 0000 0000 A200 0000 2E02 5F66
8F72 6365 7375 A200 0000 2DC4 7E7E 666F 7263 6573 8200 0000 2DC4 7E69 6D61 6765
0000 C000 0000 0003 7E73 756D 0000 0000 C000 0000 000C 7E69 0000 0000 0000 D000
0000 0007 7E77 0000 0000 0000 D000 0000 0006 7E7E 6977 0000 0000 8200 0000 2E02
7E77 7000 0000 0000 C000 0000 0003 7E77 0000 0000 0000 C000 0000 000C 7E70 0000

Drive Side Track Sektor Read Write Format Hex Ascii Exit
```

Bild 13.1 Der Diskmonitor mit Hex-Display

Das Programm reicht immerhin soweit, daß Sie damit den einfachen Kopierschutz, wie ihn speziell die ersten ST-Programme mitbekamen, aufheben können. Dazu eine Anmerkung: Ich kaufe Software, mag es aber nicht, wenn sich meine Disketten anstatt im Laufwerk auf dem Laufweg (zum/vom Hersteller) befinden. Ansonsten gibt es nur einen wirksamen Kopierschutz, nämlich ein preiswertes Programm mit guter Dokumentation.

Mit diesem Diskmonitor können Sie einen beliebigen Sektor auf der Diskette lesen und schreiben sowie jeden Track formatieren. Ein Sektor wird über die Track- und die Sektornummer angesprochen. Die Anzeige eines Sektors erfolgt wahlweise in hex oder ASCII. Im Falle »Formatieren« wird der ausgewählte Track mit soviel Sektoren formatiert, wie unter Sektornummer vorgewählt wurde.

```
Drive=0    Side=0    Track=10    Sector=2                    Operation OK

0000 000A 7E6E 756D 0000 0000 C000 FFFF FFFC 7|....~num..........~partid.......
7E69 6473 7472 0000 C000 FFFF FFF4 5F67 7061 7|~idstr.........._gpart......+.~~gp
6172 7400 8200 0000 2B2E 7E69 6D61 6765 0000 C|art.....+.~image........~pinfo..
C000 0000 000C 7E72 7061 7274 0000 D000 0000 0|......~rpart........~i..........
FFFE 7E6A 0000 0000 0000 C000 FFFF FFFC 5F73 7|..~j..............._spart......+.~~
7370 6172 7400 8200 0000 2BAE 7E69 6D61 6765 0|spart.....+.~image........~pinfo
0000 C000 0000 000C 7E72 7061 7274 0000 D000 0|........~rpart........~i........
FFFF FFFE 7E6A 0000 0000 0000 C000 FFFF FFFC 5|....~j..............._dopart......
7E7E 646F 7061 7274 8200 0000 2C2E 7E70 696E 6|~~dopart....,.~pinfo........~phy
7364 6576 C000 0000 0008 7E69 0000 0000 0000 C|sdev......~i.............~image..
C000 FFFF FDFE 7E66 6174 7369 7A00 C000 FFFF F|......~fatsiz.......~ndirs......
70F6 7E62 7300 0000 0000 C000 FFFF FDF2 5F69 7|..~bs............._iw............_f
6F72 6365 7375 A200 0000 2DC4 7E7E 666F 7263 6|orcesu....-.~~forces....-.~image
0000 C000 0000 0008 7E73 756D 0000 0000 C000 0|........~sum..........~i........
0000 0007 7E77 0000 0000 0000 D000 0000 0006 7|....~w............~~iw..........
7E77 7000 0000 0000 C000 0000 0008 7E77 0000 0|~wp...........~w............~p..

 Drive Side Track Sektor Read Write Format Hex Ascii Exit
```

Bild 13.2 Rechts das ASCII-Display als Fenster eingeblendet

Wenn das Programm startet, sehen Sie zuerst Bild 13.1, fast ... Fast deshalb, weil tatsächlich nur die erste Zeile erscheint und die letzte. Die erste Zeile ist die Statuszeile. Hier wird angezeigt, welches Laufwerk, welche Seite und welcher Track/Sektor zur Zeit gültig sind. Erst wenn Sie das Kommando R (wie Read) geben, erscheint der mittlere Teil, der hier die 256 Worte eines Sektors in hex anzeigt. Rechts in der Statuszeile erscheint dann die Anzeige »Operation OK« oder »Operation fehlerhaft«, je nachdem, ob es beim Lesen, Schreiben oder Formatieren keinen oder doch einen Fehler gab.

Unten im Bild sehen Sie die Menüzeile. Jede Funktion wird durch Eintippen des invers geschriebenen Buchstabens ausgelöst. D, I, T und S erzeugen eine Frage, zum Beispiel »Drive?« und warten auf Eingabe. Die Antwort wird dann sofort in die Statuszeile eingetragen. R und W führen einfach die entsprechende Funktion aus. Mit H oder A wird zwischen Hex- und ASII-Display hin- und hergeschaltet. Wie Bild 13.2 zeigt, wird das ASCII-Display wie ein Fenster rechts über das Hex-Display gelegt. Sobald Sie H tippen, verschwindet es wieder. Bild 13.3 bringt das Listing, im Anschluß daran kommen die Erläuterungen.

```
**************   Ein Disk-Monitor   *************

        text                        ; und schon geht es los

        bsr     init                ; Startwerte vorgeben
        bsr     status              ; Status anzeigen
        move.l  #menu,d0            ; Menu ausgeben
        bsr     print

* Hauptschleife
* ---------------------------------------------
loop

* Warte auf Taste
* --------------
        move    #7,-(sp)            ; conin ohne Echo
        trap    #1
        addq.l  #2,sp
        tst     d0                  ; kein ASCII-Zeichen?
        beq     oop                 ; wenn so
        cmpi    #'A',d0             ; kein Buchstabe?
        blt     loop                ; dann ignoriere Taste
        bclr    #5,d0               ; Erzwinge Grossbuchstaben

* Suche Tasten-Code in Tabelle
* ----------------------------
        lea     keys,a0             ; Tabelle gueltige Keys
        move    #count,d1           ; deren Anzahl
search  cmp.b   (a0)+,d0            ; Key auf aktuellem Platz?
        dbeq    d1,search           ; wenn nicht, weitersuchen
                                    ; bis Tabellenende
        tst     d1                  ; Key gefunden?
        bmi     oop                 ; wenn nicht, auf ein
                                      Neues

* Suche Adresse zu Key
* -------------------
        neg     d1                  ; sub d1,#count
        add     #count,d1           ; ergibt Platznr. von Key
        lsl     #2,d1               ; die mal 4
        lea     table,a0            ; Adr. der Routine
        move.l  0(a0,d1.w),a0       ; bestimmen
        jsr     (a0)                ; und diese aufrufen
        bra     loop                ; u.s.w.
* ---------------------------------------------------------
* Ende der Hauptschleife
```

```
* Beginn der Unterprogramme
* ---------------------------------------------------------------
print    move.l   d0,-(sp)          ; Zeile drucken
         move.w   #9,-(sp)          ; Funk. PRINTLINE
         trap     #1
         addq.l   #6,sp
         rts

;Initialisierung
;---------------
init     move     #0,driveno        ; Drive=0
         move     #0,sideno         ; Seite=0
         move     #0,trackno        ; Track=0
         move     #1,secno          ; Sektor=1
         rts

;Status ausgeben
;---------------
status   move.l   #txt_1,d0         ; Drive-Nummer
         bsr      print             ; Text 'Drive' drucken
         move     driveno,d1        ; und dann die Zahl
         bsr      prtnn

         move.l   #txt_2,d0         ; Seitennummer
         bsr      print
         move     sideno,d1
         bsr      prtnn

         move.l   #txt_3,d0         ; Track-Nummer
         bsr      print
         move     trackno,d1
         bsr      prtnn

         move.l   #txt_4,d0         ; Sektor-Nummer
         bsr      print
         move     secno,d1
         bsr      prtnn

         move.l   #prompt,d0        ; Cursor auf Prompt-Zeile
         bsr      print
         rts

;(d1) in dezi. ausgeben (<= 2 Digits)
;------------------------------------
prtnn    andi.l   #$FF,d1           ; Stellen begrenzen
         cmpi     #9,d1             ; ein Digit?
         bls      simpel            ; wenn so
         divu     #10,d1            ; Zehner ermitteln
         bsr      simpel            ; ausgeben
         swap     d1                ; Einer (Div.-Rest)
```

```
simpel   add      #48,d1            ; in ASCII
         move.w   d1,-(sp)          ; Zeichen
         move.w   #2,-(sp)          ; Funktion CONOUT
         trap     #1                ; aufrufen
         addq.l   #4,sp             ; Stack korrigieren
         rts

;Input Nummer
;------------
;Dezimalzahl wird als String gelesen und konvertiert.0
;Ergebnis in D4

input    move.b   #3,inbuff         ; max 2 Zeichen +CR lesen
         pea      inbuff            ; Pufferadresse
         move.w   #10,-(sp)         ; Funktion Zeile lesen
         trap     #1                ; aufrufen
         addq.l   #6,sp             ;
         movem.l  a0/d1-d2,-(sp)    ; benutzte Register retten
         clr.l    d1                ; Arbeitsregister initia-
                                      lisieren
         clr.l    d2
         clr.l    d4
         lea      inbuff,a0         ; Zeiger auf Puffer
         move.b   1(a0),d1          ; Anzahl Zeichen
         sub.b    #1,d1             ; -1 fuer CR
         addq.l   #2,a0             ; Zeiger auf erstes Zei-
                                      chen
30$      move.b   (a0)+,d2          ; Zeichen -> d2
         subi.b   #$30,d2           ; in Ziffer wandeln
         mulu     #10,d4            ; mal Stellenwert
         add.l    d2,d4             ; und auf Ergebnis
                                      addieren
         dbra     d1,30$            ; "for" alle Ziffern
         movem.l  (sp)+,a0/d1-d2    ; Restore Register
         move.l   #prompt,d0        ; letzte Frage loeschen
         bsr      print
         rts

;Drive/Seiten/Track/Sektor-Nummern holen
;---------------------------------------
drive    move.l   #prompt,d0        ; Cursor -> Prompt-Stelle
         bsr      print
         move.l   #txt_11,d0        ; Drive-Nr. fragen
         bsr      print
         bsr      input             ; Antwort holen
         move     d4,driveno        ; und speichern
         bsr      status            ; Statuszeile updaten
         rts
```

```
side     move.l   #prompt,d0        ; wie vor
         bsr      print
         move.l   #txt_21,d0
         bsr      print
         bsr      input
         move     d4,sideno
         bsr      status
         rts
track    move.l   #prompt,d0        ; wie vor
         bsr      print
         move.l   #txt_31,d0
         bsr      print
         bsr      nput
         move     d4,trackno
         bsr      status
         rts
sector   move.l   #prompt,d0        ; wie vor
         bsr      print
         move.l   #txt_41,d0
         bsr      print
         bsr      input
         move     d4,secno
         bsr      status
         rts
;Sektor lesen/schreiben
;----------------------
read     move     #8,f_nr           ; Entry lesen
         bsr      rdwr              ; Sektor lesen
         bsr      hex               ; und ausgeben
         rts
write    move     #9,f_nr           ; Entry schreiben
rdwr move #1,-(sp)                  ; 1 Sektor lesen/schreiben
         move     sideno,-(sp)      ; Seitennummer
         move     trackno,-(sp)     ; Track-Nummer
         move     secno,-(sp)       ; Sektor-Nummer
         move     driveno,-(sp)     ; Drive-Nummer
         clr.l    -(sp)             ; Filler;
         pea      buffer            ; Pufferadresse
         move     f_nr,-(sp)        ; Funktion (r/w)
         trap     #14               ; XBIOS aufrufen
         add.l    #20,sp            ; Stack korr.
;auf Fehler testen, OK oder Error ausgeben
errtest  tst      d0                ; Teste Status
         bne      error             ; wenn Fehler
         move.l   #ok_txt,d0
```

```
        bsr     print
        rts
error   move.l  #er_txt,d0
        bsr     print
        rts

* Formatieren eines Tracks
* ------------------------
flopfmt clr     -(sp)              ; Init-Daten
        move.l  #$87654321,-(sp)   ; Keep XBIOS happy
        move    #1,-(sp)           ; Interleave
        move    sideno,-(sp)       ; Seite
        move    trackno,-(sp)      ; Track
        move    secno,-(sp)        ; Anzahl Sektoren
        move    driveno,-(sp)      ; Laufwerk
        clr.l   -(sp)              ; Filler
        pea     buffer             ; for bad sectors
        move    #10,-(sp)          ; Funktion Format
        trap    #14                ; call it
        add.l   #26,sp             ; Stack korr.
        bra     errtest            ; teste error & ret

* Sektor als 256 Worte in Hex drucken
* Format: 16 Worte/Zeile, 16 Zeilen
* ----------------------------------
hex     move.l  #line_3,d0         ; Cursor auf Zeile 3
        bsr     print
        lea     buffer,a4
        move    #15,d1             ; 16 Zeilen zu drucken
oloop   move    #15,d3             ; 16 Worte/Zeile
iloop   clr.l   d2                 ; work register
        move    (a4)+,d2           ; 1 Wort aus Puffer
        bsr     prthex             ; drucken
        move.w  #32,-(sp)          ; Zeichen
        move.w  #2,-(sp)           ; Funktion CONOUT
        trap    #1                 ; aufrufen
        addq.l  #4,sp              ; Stack korrigieren
        dbra    d3,iloop           ; bis Zeile voll
        move.l  #crlf,d0           ; nun neue Zeile
        bsr     print
        dbra    d1,oloop           ; bis alle Zeilen
        move.l  #prompt,d0
        bsr     print
        rts

* Ein Wort in d2 in Hex (als Text) drucken
* ----------------------------------------
prthex  movem.l d0-d3,-(sp)
```

```
        move.l  #3,d1           ; loop 4 nibbles
p1      rol.w   #4,d2           ; get nibble
        move.l  d2,d3
        andi.b  #$0f,d3         ; mask it
        addi.b  #48,d3          ; '0'..'9' anyway
        cmpi.b  #58,d3          ; >9?
        bcs     out             ; no
        addi.b  #7,d3           ; 'A'..'F'
out
        move.w  d3,-(sp)        ; Zeichen
        move.w  #2,-(sp)        ; Funktion CONOUT
        trap    #1              ; aufrufen
        addq.l  #4,sp           ; Stack korrigieren
        dbra    d1,p1
        movem.l (sp)+,d0-d3
        rts

* Einen Sektor (512 Bytes) in ASCII drucken
* Format: 16 Zeilen a 32 Zeichen
* Problem: nur auf rechter Bildhaelfte
* Nicht druckbare Zeichen als Punkt
* ----------------------------------------
ascii   lea     buffer,a4
        move    #15,d1          ; 16 Zeilen zu drucken
        move.b  #32+3,d5        ; Cursor auf Zeile 3
olp     move.b  d5,TheX         ; modifiziert ESC-String!
        move.l  #line_x,d0      ; Cursor positionieren
        bsr     print
        move.w  #'|',-(sp)      ; Zeichen
        move.w  #2,-(sp)        ; Funktion CONOUT
        trap    #1              ; aufrufen
        addq.l  #4,sp           ; Stack korrigieren
        move    #31,d3          ; 32 Chars/Zeile
ilp     move.b  (a4)+,d2        ; 1 Char aus Puffer
        cmpi.b  #$1F,d2         ; nicht druckbares Zei-
                                  chen?
        ble     punkt           ; wenn ja
        cmpi.b  #$7F,d2
        ble     disp
punkt   move    #'.',d2         ; sonst '.'
disp
        move.w  d2,-(sp)        ; Zeichen
        move.w  #2,-(sp)        ; Funktion CONOUT
        trap    #1              ; aufrufen
        addq.l  #4,sp           ; Stack korrigieren
        dbra    d3,ilp          ; bis Zeile voll
        addi    #1,d5           ; nun neue Zeile
        dbra    d1,olp          ; bis alle Zeilen
```

```
        move.l  #prompt,d0
        bsr     print
        rts

exit    clr     -(sp)               ; Fall Exit
        trap    #1

        data
keys    dc.b    'D','I','T','S','R','W','F','H','A','X'
count   equ     *-keys
        ds.w    0
table   dc.l    drive
        dc.l    side
        dc.l    track
        dc.l    sector
        dc.l    read
        dc.l    write
        dc.l    flopfmt
        dc.l    hex
        dc.l    ascii
        dc.l    exit

nowrap  dc.b    27,'v',0
        ds.w    0
txt_1   dc.b    27,'H'              ; Cursor home
        dc.b    27,'K'              ; Clear to EOL
        dc.b    27,'w'              ; Wraping off
txt_11  dc.b    'Drive=',0
txt_2   dc.b    '    '
txt_21  dc.b    'Side=',0
txt_3   dc.b    '    '
txt_31  dc.b    'Track=',0
txt_4   dc.b    '    '
txt_41  dc.b    'Sector=',0

menu    dc.b    27,'Y',23+32,2+32       ; gotoyx 23,2
        dc.b    27,'pD',27,'qrive '
        dc.b    'S',27,'pi',27,'qde '
        dc.b    27,'pT',27,'qrack '
        dc.b    27,'pS',27,'qektor '
        dc.b    27,'pR',27,'qead '
        dc.b    27,'pW',27,'qrite '
        dc.b    27,'pF',27,'qormt '
        dc.b    27,'pH',27,'qex '
        dc.b    27,'pA',27,'qscii '
        dc.b    'E',27,'px',27,'qit '
        dc.b    0
        ds.w    0
prompt  dc.b    27,'Y',23+32,66+32
```

```
        dc.b    27,'K'
        dc.b    0
        ds.w    0
crlf    dc.b    13,10,0
        ds.w    0
line_3  dc.b    27,'Y',35,32,0
        ds.w    0
line_x  dc.b    27,'Y'
TheX    dc.b    32,32+46,0
        ds.w    0
er_txt  dc.b    27,'Y',32+0,32+60,27,'K'
        dc.b    'Operation fehlerhaft',0
ok_txt  dc.b    27,'Y',32+0,32+60,27,'K'
        dc.b    'Operation OK',0
        bss
driveno ds.w    1
sideno  ds.w    1
trackno ds.w    1
secno   ds.w    1
f_nr    ds.w    1
inbuff  ds.w    5
buffer  ds.b    $2000
        end
```

Bild 13.3 Listing des Diskmonitors

13.1 Menü-Technik

Die Hauptschleife kennen Sie schon aus Kapitel 5.5. Wir haben hier wieder eine Tabelle mit den erlaubten Kommandos und eine mit den zugehörigen Adressen. Neu ist, daß vorher eine Menüzeile ausgegeben wird. Diese Zeile ist nichts weiter als ein String, in dem unter anderem auch die Escape-Sequenzen für »inverse on« und »inverse off« des VT52-Emulators enthalten sind. Die GEMDOS-Funktion Nummer 9 (Print Line) gibt diese Zeichen an CONOUT weiter, und dort werden sie wie üblich als VT52-Kommandos erkannt und ausgeführt. Der Text sieht nun etwas wild aus. Wenn Sie sich aber immer vor Augen führen, daß »27,p« und »27q« den Invers-Modus ein- bzw. ausschalten, dann werden Sie die Texte wie Drive, Side, Track usw. noch erkennen.

Am Anfang der Menü-Zeile stehen zwei Escape-Sequenzen, die den Cursor so stellen, wie es ein »gotoxy« tun würde. Solche Kommandos können Sie natürlich auch an anderen Stellen im Text einbauen. Außerdem sind natürlich auch Control-Codes wie zum Beispiel »13,10« für einen Zeilenvorschub erlaubt. Langer Rede kurzer Sinn: Sie können auf diese Art beliebige Bildschirmmasken aufbauen. Solange kein Null-Byte auftritt, wird die Print-Line-Funktion alles ausgeben.

13.2 Bildschirm-Gestaltung

Prinzipiell sehen Sie in Bild 13.1 schon alle Grundelemente einer (für Programme dieser Art) typischen Bildschirm-Gestaltung. Es gibt die Statuszeile, die über den Ist-Stand aller wichtigen Parameter berichtet. Dann folgt die Ergebnis-Anzeige, hier der Inhalt eines gelesenen Sektors. Zum Schluß müssen Sie dem Bediener nur noch anzeigen, was er nun tun kann. Hier ist das die Menüzeile.

Es ist üblich, diese drei Teile in drei Unterprogramme zu verpacken. Sie sollten nun keine Hemmungen haben, diese Routinen im Programm immmer wieder aufzurufen. In Assembler geht das so schnell, daß der Benutzer das nicht merkt. Nehmen wir als Beispiel die Statuszeile. Wird hier ein Parameter geändert, zum Beispiel die Track-Nummer, dann wäre es sehr mühsam, den Cursor auf diese Position zu bringen, dort die alte Nummer zu löschen (muß sein, wenn zweistellige Nummer durch einstellige ersetzt wird), den Cursor wieder zu positionieren und nun endlich die Nummer zu schreiben. Einfacher ist es, nur den neuen Wert in die Variable Track zu schreiben und dann das Unterprogramm, das die ganze Statuszeile neu schreibt, aufzurufen.

13.3 Displays

Unser Display ist die Sektor-Anzeige, die sowohl nur in hex (Bild 3.1) als auch mit dem ASCII-Text überlagert erfolgen kann (Bild 3.2). Auch hier gilt das eben Gesagte. Es gibt zwei Routinen, nämlich »hex« und »ascii«. Wird »ascii« aufgerufen, überschreibt es einfach den rechten Teil des Hex-Displays. Wird wieder »hex« gewünscht, wird schlicht »hex« noch einmal aufgerufen. Praktisch sieht das aus wie ein Fenster, das auftaucht und wieder verschwindet. Möglich macht es Neckermann und das hohe Tempo von in Assembler geschriebenen Programmen.

Einen Trick muß ich besonders erläutern, nämlich die Cursor-Positionierung »gotoxy«, wobei X und Y Variable sein können. Da finden Sie zum Beispiel im Listing

```
        move.b    #32+3,d5      ; Cursor auf Zeile 3
olp     move.b    d5,TheX       ; modifiziert ESC-String!
        move.l    #line_x,d0    ; Cursor positionieren
        bsr       print
```

und im Datenbereich

```
line_x  dc.b      27,'Y'
TheX    dc.b      32,32+46,0
```

Die Escape-Sequenz zur Cursor-Positionierung wird durch das Unterprogramm »print« (Parameterübergabe in D0) ausgegeben. Vorher wird aber die Variable D5 (wird später geändert) in den Escape-String »line_x« eingetragen. Die dort stehende 32 wird also überschrieben.

13.4 Zugriff auf das XBIOS

Kern des Programms sind natürlich die Routinen zum Lesen, Schreiben und Formatieren von Sektoren bzw. Tracks. Hier im XBIOS sind wir schon auf der untersten Ebene des TOS. Die XBIOS-Aufrufe werden direkt in Kommandos an den FDC (Floppy Disk Controller) übersetzt. Ein Dateizugriff auf GEMDOS-Ebene hingegen resultiert im Aufruf des BIOS (Funktion rwabs, siehe Kapitel 12), und das BIOS wiederum ruft das XBIOS. Vergleichen Sie einmal »rwabs« mit dem Teil ab »Sektor lesen/schreiben«, so fällt auf, daß es hier kein »rwflag« gibt, sondern zwei getrennte Funktionen (Nummer 8 und 9). Da das der einzige Unterschied ist, unterscheide ich »read« und »write« auch nur durch unterschiedliche Werte für die Variable »f_nr« vor dem Aufruf.

Ansonsten ist die Parameter-Liste etwas anders, dürfte aber noch einleuchtend sein.

Das Formatieren eines Tracks ist da hingegen schon etwas komplizierter. Hier die ersten drei Parameter:

```
flopfmt clr     -(sp)              ; Init-Daten
        move.l  #$87654321,-(sp)   ; Keep XBIOS happy
        move    #1,-(sp)           ; Interleave
```

Der erste Parameter ist ein Testmuster, das beim Formatieren auf die Disk geschrieben und zurückgelesen wird. Ein einfaches Muster, wie hier Nullen, formatiert nahezu jeden Track, egal wie schlecht (in Grenzen) die Diskette ist. Das von Western Digital empfohlene »Pattern« heißt $E5E5. Für den Normalfall sollten Sie also die erste Zeile durch »move $E5E5,-(sp)« ersetzen.

Nun folgt die geheimnisvolle Zahl $87654321. Das soll eine Geheimzahl sein, man spricht auch von der »magic number«. Sinn der Übung ist es, daß nicht jemand aus Versehen diese Funktion mit den so schwer wiegenden Folgen aufruft. Schließlich sind nach dem Formatieren die Daten auf einer Disk unwiderruflich gelöscht.

Nun kommt »Interleave«. Der Hintergrund ist folgender: Nicht jede (kaum eine) CPU ist so schnell wie der 68000. Die »lahmen« CPU's haben also arge Probleme, wenn sie zwei Sektoren lesen sollen, die direkt hintereinander auf der Diskette liegen. Eine Folge der Art 1,2,3,4 ... schlucken sie nicht, weil sich die Diskette schon um mehr als einen Sektor weitergedreht hat, bevor die CPU die Daten übernommen hat. Damit sie nun nicht warten muß, bis der nächste Sektor mit der nächsten Umdrehung wieder vorbeikommt, gibt es den Interleave-Faktor. Dazu müssen Sie noch wissen, daß jeder Sektor einen Vorspann hat, in dem die Sektor-Nummer steht. Nun ordnet man einfach die Sektoren in der Reihenfolge von zum Beispiel 1,3,2,4... auf der Diskette an (Interleave=2) oder wählt noch größere Abstände. Beim ST brauchen wir so etwas nicht, folglich kann Interleave immer 1 sein.

13.5 Kopierschutz

Die Anzahl der Sektoren ist üblicherweise 9. 10 Sektoren sind unter Zudrücken sämtlicher Hühneraugen gerade noch möglich, allerdings schon mit dem (kleinen) Risiko von Datenverlusten.

Es gibt nun Leute, die formatieren auch noch die Tracks 80, 81 und 82, um mehr Platz zu gewinnen. Andere formatieren einen oder alle dieser Tracks, um damit einen Kopierschutz zu erzeugen. Der Gedanke dabei: Die normalen Funktionen des TOS formatieren und kopieren nur die Tracks 0 bis 79. Folglich können Daten auf zum Beispiel Track 80 nur mit spezieller Software gelesen werden, die dann natürlich Teil des »kopiergeschützten« Programms ist. Um die Sache nun noch ganz raffiniert zu machen, werden auf diesen Tracks oft nur 1 oder 2 Sektoren formatiert. Folglich nehme man den Diskmonitor und stelle fest, welche Tracks ab Nummer 80 mit wieviel Sektoren formatiert sind, sprich bis zu welchem Sektor man jeweils ohne Fehlermeldung lesen kann. Dann formatiere man eine andere Diskette entsprechend (Sektor-Nummer heißt dann Anzahl Sektoren). Das Kopieren geschieht in der Art »Sektor lesen, Drive-Nummer oder Diskette wechseln, Sektor schreiben«. Vielleicht schreiben Sie sich auch ein Programm, das diesen Vorgang automatisiert. Hier der Rahmen für einen Track:

```
        clr     d4          ; Zähler gute Sektoren
loop    add     #1,d4       ; jetzt Sektor-Nummer
        bsr     read        ; lese den Sektor
        tst     d0          ; Fehlermeldung
        beq     loop        ; wenn kein Fehler
        sub     #1,d4       ; ergibt Anzahl gute Sektoren
        bsr     format      ; damit formatieren
```

13.6 Zahlenwandlungen

Auf 2 Routinen, die Sie wahrscheinlich immer brauchen werden, möchte ich Sie noch aufmerksam machen. Im Kapitel 7 hatten wir uns ja sehr ausführlich mit »bindec« beschäftigt, der Routine zur Umwandlung einer Binärzahl in einen Dezimalstring. Wenn man Dezimalzahlen nur zweistellig ausgeben will, kann man sehr kräftig kürzen und zwar so:

```
prtnn   andi.l  #$FF,d1     ; Stellen begrenzen
        cmpi    #9,d1       ; ein Digit?
        bls     simpel      ; wenn so
        divu    #10,d1      ; Zehner ermitteln
        bsr     simpel      ; ausgeben
        swap    d1          ; Einer (Div.-Rest)
simpel  add     #48,d1      ; in ASCII
;       hier Ausgabe mit CONOUT
        rts
```

Das Prinzip von »bindec« (Division durch 10 und weiter mit dem Rest) blieb erhalten. Der Trick hier: »prtnn« ist ein Unterprogramm. Dieses Unterprogramm ruft einen Teil von sich selbst wiederum als Unterprogramm auf. Würde man dieses Teil ausgliedern, ergäbe sich diese Sequenz:

```
;       bis hierher wie oben
        bsr     simpel      ; ausgeben
        swap    d1          ; Einer (Div.-Rest)
```

```
        bsr     simpel
        rts
simpel  add     #48,d1          ; in ASCII
;       hier Ausgabe mit CONOUT
        rts
```

Der Trick erspart ein »bsr« und ein »rts«. Ob solche Tricks sauberer Programmierstil sind, will ich dahingestellt sein lassen. Tricks dieser Art sind jedoch üblich, weshalb ich sie Ihnen nicht vorenthalten möchte.

Das Gegenstück zu »bindec« ist die Routine »input«. Hier wird ein Dezimalstring eingelesen und eine Binärzahl gewandelt. Das Prinzip sieht im Kern so aus:

```
30$     move.b  (a0)+,d2        ; Zeichen -> d2
        subi.b  #$30,d2         ; in Ziffer wandeln
        mulu    #10,d4          ; mal Stellenwert
        add.l   d2,d4           ; und auf Ergebnis addieren
        dbra    d1,30$          ; "for" alle Ziffern
```

Der Gedanke dahinter ist folgender: Die Zeichen werden von links nach rechts gelesen, die Zahl 123 also in der Folge 1, 2, 3.

Jede Ziffer wird auf das Ergebnisregister (D4) addiert. Vorher wird aber D4 immer mit 10 multipliziert. Damit wandern mit jeder neuen Ziffer die in D4 schon vorhandenen Ziffern um eine Dezimalstelle nach links. Damit steht zum Schluß die zuerst gelesene Ziffer auf der höchsten Stelle, und so soll es ja wohl auch sein.

Kapitel 14

Einbindung von Assembler-Routinen
in Hochsprachen

Basic

Pascal

Die Sprache C

In den meisten Fällen ist es bequemer, Programme in einer Hochsprache zu schreiben, als sie direkt in Assembler zu formulieren. Leider ergeben sich damit dann oft genug zwei Probleme, nämlich

das Programm ist zu langsam
oder die Aufgabe ist in der Hochsprache nicht zu lösen.

Die Analyse ergibt dann, daß es immer nur eine oder wenige Stellen im Programm sind, wo es »kneift«. Den dann logischen Beschluß »dann schreibe ich das eben in Assembler« können Sie dank Ihrer erworbenen Kenntnisse jetzt fassen. Ich muß Ihnen nur noch verraten, wie man Hochsprache und Assembler miteinander verbindet.

14.1 Basic

Kümmern wir uns zuerst um Basic, hier bedarf es nämlich erfahrungsgemäß der meisten Nachhilfe durch Assembler-Routinen. Generell gibt es zwei Wege. Beiden ist gemeinsam, daß sie nur den reinen Code, wie er nach dem Assembler-Lauf (vor dem Linken) zur Verfügung steht, verwenden dürfen. Außerdem muß das Programm lageunabhängig geschrieben sein. Ein guter Assembler meldet Ihnen sogar, ob das Programm »relocatable« oder »position independent« ist. Ferner ist »Programm« eigentlich falsch, es muß ein Unterprogramm sein, das natürlich (nur mit BSR) andere Unterprogramme aufrufen darf.

14.1.1 Modul nachladen

So vorbereitet können Sie das Unterprogramm von Basic aus mit dem Befehl »BLOAD« nachladen, bleibt nur die Frage wohin. Dazu reservieren Sie mit einer DIM-Anweisung Speicherbereich. Geschah das zum Beispiel mit

```
DIM A(100)
```

so können Sie nun mit

```
X=VARPTR(A(0))
```

die Adresse feststellen, bei der der Array im RAM startet. Dahin laden Sie Ihr Modul. Die Länge müssen Sie bei BLOAD nicht angeben, jedoch sollte der Array schon so groß dimensioniert sein, daß Ihr Programm hineinpaßt. Am einfachsten rechnet es mit Integer-Arrays, dort bietet jedes Element Platz für ein Wort.

14.1.2 Routine in den RAM POKEn

Da es für den Anwender etwas unpraktisch ist, zu seinem Basic-File immer noch das passende Objekt-File auf der Diskette zu halten, wählt man meistens einen anderen Weg. Die Routine wird als Zahlenreihe in DATA-Zeilen des Basic-Programms codiert und von dort in den RAM über POKE-Befehle kopiert. Da es ziemlich mühsam ist, aus dem Listing des Assemblers den Objekt-Code abzulesen und in die DATA-Zeilen zu tippen, hier ein kleines Programm, daß Ihnen diese Arbeit abnimmt.

```
1       ' Peter Wollschlaeger's Quick & Dirty
2       ' A         B     C
3       ' Assembler Basic Converter
4       '
10      clearw 2:fullw 2
20      input"Name des Code-Files (*.o";p$
30      p$=p$+".o"
40      print "Groesse von ";p$;:input" in Bytes";b
45      b=int(b/16+0.5)*16
50      input"Name des Basic-Files (*.BAS)";b$
60      b$=b$+".BAS"
70      dim c%(b/2-1)
80      bload p$,varptr(c%(0))
90      open "O",1,b$
100     z=0:cs#=0
110     for i=1 to b/16
120     print #1,str$(i*10);" DATA ";
130     for j=1 to 8
140     print #1,right$("0000"+hex$(c%(z)),4);
150     if j<8 then print #1,",";
160     cs#=cs#+c%(z) :z=z+1
170     next j
180     print #1
190     next i
200     close #1
210     print "Die Checksum ist ";cs#
```

Bild 14.1 Ein Assembler-Basic-Konverter

Das Programm liest das Objekt-File und erzeugt daraus ein Basic-File mit den DATA-Zeilen. In diesem steht dann der Code in hex. Lesen können Sie es dann wortweise mit »read a$« und dann mit »x = val("&h"+a$)« in eine Zahl wandeln, die schließlich zu »poken« ist.

Auf eines sollten Sie in Basic immer achten. Bevor Sie mit »X=varptr(a%(0))« die Startadresse Ihres Moduls festlegen, sollten Sie alle Variablen des Programms initialisiert haben, und sei es, daß Sie Ihnen nur Blindwerte zuweisen. Wird später noch ein neuer Variablennamen gebraucht, verschiebt dieser die Variablentabelle, womit leider auch der Array mit dem Code wandert und damit dessen Startadresse.

Zur Parameterübergabe kann ich mich nicht detailliert äußern, weil sich hier die Basic-Dialekte nicht einig sind. Die sicherste Methode ist der Transfer über die »tote Ecke« im Video-RAM. Für das Video-Ram sind $8000 Bytes reserviert, was für einen Assembler-Programmierer 32 K sind oder 32768 Bytes. Tatsächlich braucht der ST aber nur 32000

Bytes davon, 768 sind also immer frei. Für die Parameterübergabe reicht das immer (bei Arrays übergibt man nur Adressen).

14.2 Pascal

Hier behandle ich nur ST-Pascal, wer einen anderen Dialekt bevorzugt, sollte in dessen Handbuch nachschlagen. Am einfachsten schildert man die Sache an einem Beispiel, der Einfachheit halber gleich an dem Problem »Track/Sektor lesen« aus dem vorigen Kapitel. Das Pascal-Programm deklariert dafür nur eine Funktion als »external«. Dieser Funktion werden die nötigen Parameter übergeben. Die Funktion soll dann die Adresse des Puffers mit den Daten zurückgeben. Bild 14.2 zeigt die Details.

```
{$S10}
program readts;
type buffer = packed array[1..257] of integer;
     bufptr = ^buffer;
var bp: bufptr;
function RDTS(Drive,Track,Sector: integer): bufptr;
  EXTERNAL;
begin
  bp:= RDTS(0,3,1);
  write(bp^[1]);
end.
```

Bild 14.2 Pascal ruft Assembler

Die Assembler-Routine selbst zeigt Bild 14.3. Sie holt zuerst die Return-Adresse vom Stack und dann die Parameter. Dazu muß man sich nur merken, daß man die Parameter in der umgekehrten Reihenfolge vom Stack holen muß, wie man Sie in der Funktion genannt hat. Das hat Pascal mit C gemeinsam. Beachten Sie bitte, daß ich zuerst die Return-Adresse rette. Dieser Schritt ist empfehlenswert, weil so – was auch immer mit dem Stack passiert – der Rücksprung gesichert ist.

```
****** Drive, Track, Sektor lesen
        xdef      RDTS

RDTS    move.l    (sp)+,retsave ; save ret adr
        move.w    (sp)+,d2      ; get sector
        move.w    (sp)+,d1      ; get track
```

```
        move.w    (sp)+,d0      ; get drive

        move.w    #1,-(sp)      ; 1 Sektor
        move.w    #0,-(sp)      ; Seite 0
        move.w    d1,-(sp)      ; Track
        move.w    d2,-(sp)      ; Sektor
        move.w    d0,-(sp)      ; Drive
        clr.l     -(sp)         ; Filler
        pea       buffer        ; Bufferadresse
        move.w    #8,-(sp)      ; Funktion Read T/S
        trap      #14           ; call XBIOS
        add.l     #20,sp

        move.w    d0,flag       ; Status uebergeben
        move.l    #buffer,d0    ; Pufferadresse als Ergebnis
        move.l    retsave,a0    ; get ret adr
        jmp       (a0)          ; ret to pascal

        bss
retsave ds.l      1             ; Return Adresse
buffer  ds.w      256           ; Puffer Sektor Daten
flag    ds.w      1             ; Status (d0) des Read
        end
```

Bild 14.3 Die von Pascal gerufene Routine

Da eine Funktion immer nur einen Wert zurückgeben kann (hier die Pufferadresse), ich aber noch eine eventuelle Fehlermeldung nicht verpassen möchte, kommt jetzt ein kleiner Trick. Das Register D0 wird auf das Label unter »Flag« gesichert und ist damit das 257. Element im Pascal-Array. Auf diese Art lassen sich beliebig viele Daten zurückgeben.

Zum Schluß werden beide Routinen (Pascal compiliert, Assembler assembliert) einfach mit dem Linker gebunden. Der Linker will natürlich informiert sein. Das geschieht so: Die externe Funktion in Pascal heißt RDTS, auch RDTS heißt die Assembler-Routine. Mit »xdef RDTS« wird dieser Name dem Linker bekanntgemacht.

14.3 Die Sprache C

Generell kann man in C so vorgehen, wie eben in Pascal geschildert, doch gute C-Compiler machen dem Programmierer das Leben noch leichter, indem sie sogenannten Inline-Code erlauben, wie es zum Beispiel Megamax tut. Bild 14.4 zeigt ein Beispiel.

```
#include <osbind.h>
long *scrbase;
main()
{
    register int i,j;
    scrbase = (long *) Logbase();
    for (j = 1; j <= 50; j++)
      asm
      {
           move.l  scrbase(A4), A0
           move.w  #8000-1, D0
      lp:  not.l   (A0)+
           dbf     D0,lp
      }
}
```

Bild 14.4 Assembler-Einbindung in C: einfacher geht's nicht

Das kleine Beispiel invertiert den halben Video-RAM und das in der FOR-Schleife 50mal. Da das sehr schnell geht, sieht der Anwender nur kurzzeitig ein sehr schnelles Bildschirmflackern. Die Assembler-Routine kann sehr einfach C-Variable benutzen. Globale und statische Variable werden als Offset zu A4 angegeben, die anderen als Offset zu A6. Für Registervariable kann man direkt deren Namen heranziehen. Wenn Sie das Programm einmal in Assembler ausprobieren möchten, schreiben Sie

```
        move      #2,-(sp)
        trap      #14          ; get screen base
        addq.l    #2,sp
        move.w    #8000-1, D0
lp:     not.l     (A0)+
        dbf       D0,lp
```

Der Sprachschatz des Megamax-Assemblers ist etwas eingeengt. So mag er zum Beispiel nicht »dbra«, sondern kennt nur »dbf«. Auch müssen Registernamen generell groß geschrieben werden. A7 kennt er, SP nicht. Ansonsten ist die maschinennahe Sprache C Assembler am nächsten. Falls Sie C noch lernen wollen, so haben Sie mit diesem Assembler-Studium die richtige Basis geschaffen. Ich persönlich meine, so sollte es sein: erst Assembler, dann C. Einige Leute verwechseln da etwas. In C kann man zwar maschinen-unabhängig programmieren, aber nicht »maschinen-unwissend«.

Kapitel 15

Die Tricks der Profis

Einiges aus der ST-Software

So startet der ST

Cartridge-Programm

Trap-Dispatcher

Nachdem man die Grundregeln der Assembler-Programmierung gelernt hat, gibt es zwei Dinge, die man tun sollte, um sich zu perfektionieren. Das eine ist Üben, Üben, Üben ..., das andere ist Lernen, Lernen, Lernen ....

Letzteres können Sie zum Beispiel auf Kursen, die meistens »Programmer's Workshop« oder ähnlich heißen, aber diese Kurse sind meistens sehr teuer. Wenn mich meine Firma nicht hinschicken würde, privat würde ich sie nicht bezahlen. Andererseits, die Firmen zahlen die hohen Teilnehmergebühren natürlich, weil die Kurse etwas bringen. Wenn Sie also die Gelegenheit haben, nutzen Sie sie.

Ein anderer Weg ist das, was die Amerikaner so schön »steel from the best« (stehle vom Besten) nennen. Das heißt, schauen Sie sich an, wie andere Leute programmieren, die besser sind als Sie und auch deutlich besser als der Durchschnitt. Profis verraten normalerweise ihre Tricks nicht, mit einer Ausnahme: Sie veröffentlichen etwas in einer Fachzeitschrift, die für ihr hohes Niveau bekannt ist. Ein gutes Beispiel ist Dr. Dobb's Journal, das inzwischen auch in Deutschland vertrieben wird. Auch vielversprechend ist die amerikaniche Zeitschrift »STart« (ST Kunst), bei der mir die Autorenliste positiv auffiel. Dafür schreibt zum Beispiel ein Mann, der bei Digital Research das GEM mitentwickelt hat.

Ansonsten ist aber in Ihrem ST eine riesige Sammlung professioneller Listings quasi schon eingebaut. Wenn Sie einmal etwas aus dem ROM oder dem TOS nach dem Laden von der Diskette disassemblieren, stoßen Sie auf diese Quellen. Der Zugang ist an sich recht einfach. Zum einen steht in der Systemvariablen $4F2, wo das TOS im RAM oder ROM startet, zum anderen kommen Sie natürlich über die Trap-Vektoren schnell zum Ziel.

15.1 So startet der ST

Bild 15.1 zeigt ganz grob, was der ST alles tut, wenn Sie ihn »anwerfen«. Die Adressen beziehen sich noch auf eine Disk-Version, im ROM startet das Programm bei $FC0000. Ich will nun gar nicht auf Einzelheiten eingehen, sondern Ihre Aufmerksamkeit auf die unterstrichenen Teile lenken.

Der ST hat bekanntlich links den Cartridge-Slot, in den ein Modul mit Software im ROM oder EPROM eingesteckt werden kann. Wer seine eigene Software da sicher unterbringen will, hat gute Karten. Man braucht tatsächlich nur eine Karte mit 4 EPROMS des Typs 27256. Alle Signale für deren Betrieb incl. Versorgungsspannung liegen schon am Slot an. Das Schaltbild der Karte wird damit so trivial, daß ich mir das hier wohl schenken kann, zumal schon diverse Firmen solche Karten anbieten.

```
$6100    BRA    BIOS
         einige Konstanten
BIOS     Sound-Chip auf Ausgabe
         Floppies deselektieren
         Farbpalette laden
```

```
        Speicher bis Ende löschen
        Speicherende-32K=Video-RAM
        Systemvariable und SP laden
$6252   CMP.L  #$FA52235F,Ca_Base
        ------------------------
        BEQ   $62D6
        Exeption-Vektoren laden
        VBL-Liste löschen
        MPF initialisieren
$62D6   MOVEQ.L  #2,D0
        BSR   Ca_Test
        -------------
        Monitor testen
        Auflösung anpassen
        Video initialisieren
        VBL freigeben
$633E   CLR    D0
        BSR   CA_Test
        -------------
        Interrupts frei
$6348   MOVEQ.L  #1,D0
        BSR   Ca_Test
        GEMDOS initialisieren
        BSR  Diskboot
        BSR   Auto (and exec if)
        GEMDOS aufrufen
Diskboot MOVEQ.L  #3,D0
        BSR   Ca_Test
        --------------
        booten von Disk
        RTS
Ca_Test ..... siehe Bild 15.2
```

Bild 15.1 So startet der ST

15.2 So muß ein Cartridge-Programm aussehen

Damit Sie nun – auch ohne EPROM – etwas Nutzen aus der Sache ziehen können, möchte ich Ihnen zeigen, was bei »Ca_Test« passiert. Diese Routine wurde auch aus dem TOS disassembliert und dann in einen Rahmen gepackt, damit man sie praktisch ausprobieren kann. Der Rahmen beschreibt ansonsten, wie man ein Cartridge-Programm schreiben muß und dient gleichzeitig noch als Testumgebung für Cartridge-Programme, bevor man sie in den EPROM »schießt«.

```
*CA Cartridge-Programm simulieren

;Drei schon bekannte Makros
;--------------------------
conin   macro
        move.w  #1,-(sp)          ; Funktion CONIN
        trap    #1
        addq.l  #2,sp
        endm

term    macro
        clr     -(sp)             ; Funktion TERM
        trap    #1
        endm

writes  macro
        pea     \1
        move.w  #9,-(sp)          ; Funk. PRINTLINE
        trap    #1
        addq.l  #6,sp
        endm

*
* Start hier
*

loop    writes  query
        conin                     ; warte auf Taste
        cmp     #'x',d0           ; Abbruch?
        beq     fini              ; wenn so
        sub     #'0',d0           ; in Ziffer wandeln
        and.l   #%1,d0            ; nur Bit 0 gueltig
        jsr     ca_test           ; Ca-Programm aufrufen
        bra     loop              ; auf ein Neues

fini    term

;Die folgende Routine ist aus dem ST-BIOS disassembliert.
;Disk-TOS Version 1 vom 20.11.85
;-------------------------------------------------------

ca_test lea     ca_base,a0        ; Cartridge Basis-Adresse
        cmp.l   #$ABCDEF42,(a0)+  ; Teste Kennwort und
                                  ; zeige auf ersten Entry
        bne     return            ; Ende, wenn kein
                                    Cartridge
                                  ; installiert
ca1     btst    d0,4(a0)          ; Teste Init-Bit
        beq     noinit            ; wenn nicht gesetzt
```

```
        movem.l d0-d7/a0-a6,-(sp) ;  Register retten
        move.l  4(a0),a0          ; Adresse der Routine
        jsr     (a0)              ; Aufruf
        movem.l (sp)+,d0-d7/a0-a6 ; Register zurueck
noinit  tst.l   (a0)              ; gibt es noch ein Pgm?
        move.l  (a0),a0           ; wir tun mal so
        bne     cal               ; wenn ja
return  rts                       ; sonst Ende

;Die folgenden Zeilen simulieren ein Cartridge mit 2 Pro-
 grammen
;-----------------------------------------------------------
*****   org     $FA0000           ; Falls Sie einen EPROM
                                    haben

ca_base dc.l    $ABCDEF42         ; Das Kennwort
        dc.l    pgm_2             ; Zeiger auf naechsten
                                    Header
        dc.l    init_1+$01000000  ; Adresse und Bit 24
                                    gesetzt
        dc.l    start_1           ; Startadresse des ersten
                                    Pgm
        dc.w    0                 ; Uhrzeit
        dc.w    0                 ; und   Datum, wenn Sie
                                    wollen
        dc.l    end_1-start_1     ; Laenge des Pgm
        dc.b    'filename.ext',0  ; File-Name

;next header
;----------
pgm_2   dc.l    0                 ; 0 = kein weiteres Pgm
        dc.l    init_2+$02000000  ; Jetzt Adresse mit Bit 25
        dc.l    start_2           ; Adresse des Pgm

;       wie vorher                ; kann hier auch entfallen

;-----------------------------------------------------------
;hier starten die Programme mit ihren Init-Routinen

init_1  writes  msg1
        rts

start_1 rts
end_1   equ     *

init_2  writes  msg2
        rts

start_2 rts
end_2   equ     *

query   dc.b    13,10,'Enter      digit ',0
```

```
        ds.w    0
msg1    dc.b    ' -> Ca-Pgm One Initialized',0
        ds.w    0
msg2    dc.b    ' -> Ca-Pgm Two Initialized',0
        end
```

Bild 15.2 Ein Cartridge-Simulator

In Bild 15.1 sind einige Stellen durch Unterstreichungen herausgehoben. Hier wird mehrmals Service für Cartridge-Programmierer geboten. Was das bedeutet, sehen Sie am besten in Bild 15.2, das Sie zuerst ab Label ca_test betrachten sollten. Danach sollte jedes ROM-Modul mit einem Kennwort des Inhalts $ABCDEF42 starten. Die Routine prüft nämlich, ob sich diese »magic number« auf der Adresse $FA0000 befindet, was nur sein kann, wenn da a) ein Modul eingesteckt und b) »magic< gültig ist. Andernfalls geht es sofort zum Return. Nun kommt der Test von D0, und hier steckt der casus knacksus, ein ganz raffinierter Trick.

Im übernächsten Langwort (also ab $FA0008) muß die Adresse einer Routine stehen, die das ROM-Programm ggf. initialisiert. Da die Adresse nur 24 Bit belegt, bleiben 8 frei, und diese werden ganz genial zur Parameter-Übergabe benutzt, wie ich gleich noch zeigen werde.

Ab Label ca_Base steht nun, wie der Header eines ROM-Programms maximal aussehen muß, im Minimum reichen auch die ersten 3 Zeilen. Die Sache ist deshalb so kompliziert, weil zweierlei Luxus geboten wird. Zuerst: es dürfen mehrere Programme im EPROM sein (128 K in Maschinen-Code ist schließlich allerhand Holz). Zweitens: Der Programmierer kann wählen, ab wo im Boot-Prozeß des ST er sich einklinken will. Und sozusagen als Zugabe: Alle Programme im EPROM können zuerst nur initialisiert werden. Danach kann dann gar nichts passieren oder das letzte Programm in der Kette ruft nach der Initialisierung eines der Programme im EPROM auf. Falls nichts (außer der Init-Routine) passiert, dann wird wohl ein Programm im RAM das Modul-Programm aufrufen, womit auch für Brocken über 128 K ein recht wirkungsvoller Kopierschutz möglich ist.

Jedes (oder auch nur das einzige) Programm im EPROM muß mit einem Header (Vorspann) starten. Im Falle vieler kleiner Programme sind die Header als einfach verkettete Liste gestaltet. Der erste Eintrag in einem Header ist der Zeiger auf den Header des nächsten Programms oder Null, wenn keines mehr folgt. In dem kleinen Beispiel habe ich 2 Header und damit zwei Programme simuliert. Die Link des zweiten ist NIL (Null), womit die Liste ordentlich abgeschlossen ist.

Das BIOS testet nun an verschiedenen Stellen, ob sich ein Cartridge-Programm jetzt einklinken will.

Dazu steht da die Folge

```
        move    #2,d0
        jsr     ca_test
```

In D0 ist immer nur ein Bit von Bedeutung (in diesem Beispiel Bit 1). Das zu testende Bit kommt in D0 zwar unten (Bit 0..7) an, der BTST-Befehl mit einer Speicheradresse als Zieloperanden

```
        btst    d0,4(a0)
```

testet aber immer nur das Bit im Byte ab Zieladresse. Folglich korrespondiert Bit 0 in D0 mit Bit 24, Bit 1 mit Bit 25 u.s.w. Entsprechend der gesetzten Bits in den Adressen der Routinen spricht der Test dann an oder nicht. Folglich kann man die beiden Routinen, in deren Adressen ich Bit 24 bzw. 25 gesetzt habe, durch entsprechende Werte in D0 erreichen.

Im Listing von Bild 15.2 ist dieses als Eingabe realisiert. In einer Endlosschleife wird ein Zeichen von der Tastatur geholt, wobei ich unterstelle, daß Sie auch nur x für Ende oder eine Ziffer eingeben. Nur 0 oder 1 (bzw. 8 oder 9) dürften dann die jeweilige Routine aktivieren.

Lt. Bild 15.1 dürfte wohl der sicherste Weg sein, Bit 27 zu setzen.

Bliebe noch die »magic« number $FA522235F zu erwähnen. Die sollten Sie nicht nehmen, weil an dieser das System noch so wenig initialisiert ist, daß Ihnen verdammt viel Arbeit bleibt. Wie ich gehört habe, wird damit der Diagnose-ROM identifiziert. Auch nur aus zweiter Hand und nicht überprüft ist die Aussage, daß bei Desk-Programmen Bit 29 gesetzt werden muß.

15.3 Der Trap-Dispatcher des BIOS

Den Trap-Dispatcher habe ich für Sie disassembliert, weil hier sehr schön demonstriert wird, wie sauber das BIOS programmiert ist, und wie sehr hier auf Datenkonsistenz und Zuverlässigkeit geachtet wird. Ich habe noch nicht – im Gegensatz zu so manchen Schreibern – entdecken können, daß das TOS unzuverlässig ist. Ich kenne nur unzuverlässige Anwender-Programme.

Schauen wir uns mit Bild 15.3 einmal den Ist-Stand an, und nun sage mir noch jemand, was man da besser machen kann (Schneller werden durch Weglassen gilt nicht)!

```
* TDISP Demo Trap-Dispatcher

;Um den Dispatcher zu testen, wird Trap #7 benutzt
;-------------------------------------------------
        pea     tdisp           ; Adresse des Dispatchers
        move    #39,-(sp)       ; Vektor 39 = Trap  #7
        move    #5,-(sp)        ; BIOS-Funktion SETEXCEP
        trap    #13             ; aufrufen
        addq.l  #8,sp
```

```
;nun folgen 3 Aufrufe
;-------------------
        move    #'A',-(sp)          ; wir simulieren
        move    #2,-(sp)            ; conout
        trap    #7
        addq.l  #4,sp

        move    #1,-(sp)            ; und nun das neue conin
        trap    #7
        addq.l  #2,sp

        clr     -(sp)               ; und schliesslich term
        trap    #7

;hier startet eine Kopie des Trap-Dispatchers des ST
;im ROM-TOS ab $FC074E
;-----------------------------------------------------
tdisp
        lea     table(pc),a0        ; Adressen der Routinen
        move.l  #save,a1            ; Datensicherungsbereich
                                    ; im Original $4A2
        move    (sp)+,d0            ; SR holen
        move    d0,-(a1)            ; und retten
        move.l  (sp)+,-(a1);        ; PC holen und retten
        movem.l d3-d7/a3-a7,-(a1)   ; und Register retten
        move.l  a1,save             ; Zeiger retten
        btst    #13,d0              ; Aufruf im Supervisor-
                                      Modus?
        bne.s   super               ; wenn so
        move.l  usp,sp              ; benutze USP
super   move    (sp)+,d0            ; hole Funktionsnummer
        cmp     (a0)+,d0            ; vergleiche mit Count
        bge.s   exit                ; Abbruch wenn groesser
        lsl     #2,d0               ; Funkionsnummer mal 4
        move.l  0(a0,d0),d0         ; Adresse der Routine
        move.l  d0,a0               ; auch noch nach a0
        bpl.s   direct              ; wenn kein Vektor
                                      (Bit 31=0)
        move.l  (a0),a0             ; sonst ARI
direct  sub.l   a5,a5               ; a5 loeschen
        jsr     (a0)
exit    move.l  save,a1             ; auf Sicherungsbereich
        movem.l (a1)+,d3-d7/a3-a7   ; Register zurueck
        move.l  (a1)+,-(sp)         ; PC wieder auf Stack
        move    (a1)+,-(sp)         ; SR dto.
        move.l  a1,save             ; Zeiger zurueck
        rte
```

```
;hier werden 3 Routinen simuliert
;--------------------------------
table   dc.w    3                   ; Anzahl der Routinen
        dc.l    term                ; Adress-Tabelle
        dc.l    conin
        dc.l    conout

term    move.l  (sp)+,(sp)
        clr     -(sp)
        trap    #1

conin   move    #1,-(sp)
        trap    #1
        addq.l  #2,sp
        rts

conout  move    4(sp),-(sp)
        move    #2,-(sp)
        trap    #1
        addq.l  #4,sp
        rts

        ds.l    20
save    equ     *

        end
```

Bild 15.3 Der Trap-Dispatcher des BIOS

Wiederum habe ich einen kleinen Rahmen um die Routine gelegt, um Sie testen zu können. Dazu wurde auf die in Kapitel 11 vorgestellte Weise der Trap Nummer 7 auf diese Routine gestellt.

Dann werden drei Routinen zur Verfügung gestellt, die Sie wie TERM, CONIN und CONOUT des GEMDOS aufrufen können, allerdings mit jetzt Trap #7. Der Dispatcher selbst startet bei »tdisp«. An dieser Stelle ist auch der Eingang für den Trap #13. Trap #14 benutzt die Routine auch, der einzige Unterschied ist, daß bei seinem Entry »lea XBIOS-Tabelle, a0; bra tdisp + 1 Zeile«. Interessant ist nun, daß der Dispatcher über einen eigenen – als Stack organisierten – Datensicherungsbereich verfügt, wo als erstes das Status-Register, der PC und D3-D7 sowie A3-A7 gesichert werden. Weil all diese Register zum Schluß zurückgeschrieben werden, ist auch der Stackpointer wieder im Zustand wie vor dem Aufruf. Hier liegt also der Grund für das obligatorische »add.l #n,sp« nach jedem Trap.

Interessant sind nun die berühmten Kleinigkeiten. So wird zum Beispiel das Supervisor-Bit getestet, um festzustellen, woher der Aufruf kam. Schließlich müssen die Parameter vom richtigen Stack geholt werden. Der nächste Trick steckt in einem Test auf Bit 31 (bpl) der gerade ermittelten Adresse der Routine. Einige Adressen sind nämlich keine, sondern nur Vektoren (die, die wir im RAM-Disk-Programm benutzt haben). In diesen Vektoren ist

Bit 31 gesetzt. Das kann man als Vorzeichen ansehen, somit mit »bpl« testen. Ist das Bit gesetzt, wird durch »move.l (a0),a0« die Adresse indirekt benutzt.

Nun fragen Sie vielleicht noch, warum mit »sub.l a5,a5« dieses Register auf Null gesetzt wird. Schauen wir uns dazu eine einfache Routine an, nämlich die, die die logische Video-Adresse holen soll (Systemvariable $44E):

```
move.l   $44E(a5),d0
rts
```

Lassen wir das »(a5)« weg, funktioniert die Routine auch. Der Code wird nicht länger (es bleibt bei zwei Worten) und 16 Taktzyklen verbrauchen beide Adressierungsarten. Also, was soll's? Nun, es ist ein Hintertürchen, daß sich clevere Programmierer immer offen lassen, zumal wenn es – wie hier – nichts kostet. Die Systemvariablen könnten ja mal in einen anderen Adreßbereich verschoben werden. Dann müßte man bei absoluter Adressierung sehr viel ändern.

Letzter Hinweis:

Sie sehen, alle Funktionen werden als Unterprogramm aufgerufen. Die Funktionsnummer wurde aber schon vorher vom Stack genommen, bleibt die Return-Adresse. Daher haben Sie auf den letzten Parameter Zugriff mit »4(sp)«.

Letzter Tip:

Im Falle eines Crashs stehen im Datensicherungsbereich eine Menge wertvoller Informationen, nämlich aus dieser Folge

```
move     (sp)+,d0            ; SR holen
move     d0,-(a1)            ; und retten
move.l   (sp)+,-(a1);        ; PC holen und retten
movem.l  d3-d7/a3-a7,-(a1)   ; und Register retten
```

Wenn Sie nun einmal ein Programm schreiben, das diese Daten in einen anderen Bereich kopiert, bevor der nächste Trap-Aufruf erfolgt, und sie dann ausgibt, dann haben Sie ein recht nützliches Debug-Tool. Lassen Sie das Programm zuerst die Fehler-Vektoren auf sich »verbiegen« und halten Sie es dann resident, wie bei der RAM-Disk gezeigt.

Anhang

A1
Befehlsliste des 68000
A2
GEMDOS-Funktionen
A3
BIOS-Funktionen
A4
XBIOS-Funktionen
A5
LINE-A-Grafik
A6
Der Zeichensatz des Atari-ST
A7
Der VT52-Emulator
A8
Die Systemvariablen
A9
Liste der Exception-Vektoren des ST

Anhang A1
Befehlsliste des 68000

Die Befehle sind in einem kompakten Format wie im folgenden Beispiel dargestellt:

ADD ea,Dn / ADD Dn,ea — Addiere
B W L — S+D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	WL	*	*	*	*	*	*	*	*	*
D:			*	*	*	*	*	*	*			

In der ersten Zeile steht immer der Befehl in den erlaubten Syntax-Formen, gefolgt von einer kurzen Erklärung.

Dn ist ein beliebiges Datenregister.
An ist ein beliebiges Adreßregister.
Rn ist ein beliebiges Adreß- oder Datenregister.
S ist der Source-(Quell) Operand.
D ist der Destination-(Ziel) Operand.

#K ist eine Konstante
d ist die Adreßdistanz

In der zweiten Zeile stehen die erlaubten Operandengrößen (B, W, L). Darunter stehen die möglichen Adressierungsarten. Ein »*« heißt, daß alle vorgenannten Operandengrößen auch bei dieser Adressierung erlaubt sind. Ein oder zwei Buchstaben beschränken die Adressierungsart auf die durch die Buchstaben angedeuteten Operandengrößen.

cc in zum Beispiel DBcc heißt »Condition Code«. Seine Bedeutung ist auf der letzten Seite dieses Anhangs aufgeführt.

ABCD Dn,Dn / ABCD -(An),-(An) — Addiere BCD
B — S+D+X —> D

ADD ea,Dn / ADD Dn,ea — Addiere
B W L — S+D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*		WL	*	*	*	*	*	*	*	*	*	*
D:*			*	*	*	*	*	*	*			

ADDA ea,An
W L

Addiere Adresse
S+D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*		*	*	*	*	*	*	*	*	*	*	*

Wortoperand wird wie bei EXT.L erweitert

ADDI #K,ea
B W L

Addiere Konstante
#K+D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*			*	*	*	*	*	*	*			

ADDQ #K,ea
B W L

Addiere Konstante Quick (#K <=8)
#K+D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		WL	*	*	*	*	*	*	*			

ADDX Dn,Dn / ADDX -(An),-(An)
B W L

Addiere mit X-Flag
S+D+X —> D

AND ea,Dn / AND Dn,ea
B W L

Logisch UND
S AND D —> D

	DN	AN	(AN)	(AN)+	-(AN)	D(AN)	D(AN,RN)	$.W	$.L	d(PC)	d(PC,RN)	#
S:*			*	*	*	*	*	*	*	*	*	*
D:			*	*	*	*	*	*	*			

ANDI #K,ea
B W L

Logisch UND mit Konstante
#K AND D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

ANDI #K,CCR
B

Unde zu CCR
#K AND CCR —> CCR

ANDI #K,SR
W

Unde zu SR ! Privilegiert !
#K AND SR —> SR

ASL Dn,Dn / ASL #K,Dn / ASL ea
B W L

Arithmetisch links schieben
D n Bits geschoben —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			W	W	W	W	W	W	W			

0 wird nachgeschoben, herausgeschobenes Bit geht in das C- und X-Flag

---

ASR Dn,Dn / ASR #K,Dn / ASR ea
B W L

Arithmetisch rechts schieben
D n Bits geschoben —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			W	W	W	W	W	W	W			

MS-Bit schiebt, bleibt aber erhalten. Herausgeschobenes Bit geht in das C- und X-Flag

---

Bcc Label
.B W
.S

Verzweige wenn cc (PC-relativ)
siehe cc-Tabelle
PC+d —> PC

BCHG Dn,ea / BCHG #K,ea
B L

Bit n Testen und ändern
Bit-Test —> Z-Flag
Bit ändern

Wenn Source Dn: n=0..31, sonst 0..7. Wenn Destination im RAM, wird immer 1 Byte gelesen und n=n mod 8.

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:L			B	B	B	B	B	B	B			

BCLR Dn,ea / BCLR #K,ea — Bit n Testen und Löschen
B L — Bit-Test —> Z-Flag
Bit = 0

Wenn Source Dn: n=0..31, sonst 0..7. Wenn Destination im RAM, wird immer 1 Byte gelesen und n=n mod 8.

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:L		B	B	B	B	B	B	B			

BRA Label — Verzweige zu Label (PC-relativ)
.B W — PC+d —> PC
.S

BSET Dn,ea / BSET #K,ea — Bit n Testen und Setzen
B L — Bit-Test —> Z-Flag
Bit = 1

Wenn Source Dn: n=0..31, sonst 0..7. Wenn Destination im RAM, wird immer 1 Byte gelesen und n=n mod 8.

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:L		B	B	B	B	B	B	B			

BSR Label — Call Sub bei Label (PC-relativ)
.B W — PC —> -(SP); PC+d —> PC
.S

BTST Dn,ea / BSET #K,ea — Bit n Testen
B L — Bit-Test —> Z-Flag

Wenn Source Dn: n=0..31, sonst 0..7. Wenn Destination im RAM, wird immer 1 Byte gelesen und n=n mod 8.

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:L		B	B	B	B	B	B	B	B	B	

CHK ea,Dn — Register gegen Limits checken
if Dn <0 or Dn >(ea) then trap

W

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:W		W	W	W	W	W	W	W	W	W	W

CLR ea — Lösche Operand
B W L — 0 —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

CMP ea,Dn — Vergleiche Operanden
B W L — Flags wie nach D minus S

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	WL	*	*	*	*	*	*	*	*	*	*

CMPA ea,An — Vergleiche Adressen
W L — Flags wie nach D minus S

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	*	*	*	*	*	*	*	*	*	*

Wort-Operand wird vorher auf Long erweitert

CMPI #K,ea — Vergleiche gegen Konstante
B W L — Flags wie nach D minus S

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

CMPM (An)+,(An)+ — Vergleiche Speicherstellen
B W L — Flags wie nach D minus S

DBcc Dn,Label — Teste cc. Dekrementiere Dn. Branch
if cc = false then Dn=Dn-1
if Dn <> -1 then BRA Label
else »hier weiter«

DIVS ea,Dn — Dividiere Worte Signed
W — D/S —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

Quotient im niederwertigen Wort, Rest im höherwertigen

DIVU ea,Dn — Dividiere Worte Unsigned
W — D/S —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

Quotient im niederwertigen Wort, Rest im höherwertigen

EOR Dn,ea — Logisch XOR
B W L — S xor D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

EORI #K,ea — Logisch XOR mit Konstante
B L — S xor D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

EORI #K,CCR — XOR Konstante mit CCR
B — S xor CCR —> CCR

EORI #K,SR — XOR Konstante mit SR !Privileg. !
W — S xor CCR —> CCR

EXG Rn,Rn — Tausche Register
L — Rn <—> Rn

EXT Dn — Dn vorzeichenrichtig erweitern
W L

ILLEGAL — löst Illegal-Exception aus

JMP ea — absoluter Sprung (lang)
D —> PC

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

JSR ea

absoluter UP-Aufruf
PC —> -(SP); D —> PC

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

LEA ea,An
L

Lade effektive Adresse
D —> An

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

LINK An,#d

Lokalen Stack einrichten
An —> -(SP); SP —> An; SP+d —>SP

LINK und UNLK werden gebraucht, um eine »linked list« von lokalen Variablen für verschachtelte UP-Aufrufe anzulegen

---

LSL Dn,Dn / LSL #K,Dn / LSL ea
B W L

Logisch links schieben
D n Bits geschoben —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			W	W	W	W	W	W	W			

0 wird nachgeschoben, herausgeschobenes Bit geht in das C- und X-Flag

---

LSR Dn,Dn / LSR #K,Dn / LSR ea
B W L

Logisch rechts schieben
D n Bits geschoben —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			W	W	W	W	W	W	W			

0 wird nachgeschoben, herausgeschobenes Bit geht in das C- und X-Flag

---

MOVE ea,ea
B W L

Kopiere Daten
D —> S

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*		WL	*	*	*	*	*	*	*	*	*	*
D:*			*	*	*	*	*	*	*			

MOVE ea,CCR — CCR laden
W — ea —> CCR

Zwar ist das CCR nur 8 Bit breit, bei Bewegung eines Wortes in das CCR wird die obere Worthälfte jedoch ignoriert.

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*			*	*	*	*	*	*	*	*	*	*

MOVE ea,SR — SR laden ! Privilegiert !
W — ea —> SR

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*			*	*	*	*	*	*	*	*	*	*

MOVE SR,ea — SR holen ! Privilegiert !
W — SR —> ea

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*			*	*	*	*	*	*	*			

MOVE USP,An MOVE An,USP — USP holen und laden ! Privilegiert !
L — USP —> An

MOVEA ea,An — Kopiere Adresse
WL — ea —> An

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*		*	*	*	*	*	*	*	*	*	*	*

MOVEM R_Liste,ea / MOVEM ea,R_Liste — Register-Liste kopieren
W L

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
1:			*		*		*	*	*			
2:			*	*		*	*	*	*	*	*	

1= Register —> Speicher z. B.: movem d1-d3/a1-a4,-(a7)
2= Speicher —> Register z. B.: movem (a7)+,d1-d3/a1-a4

MOVEP Dn,d(An) / MOVEP d(An),Dn — Daten von / zu Peripherie
W L
Daten werden byteweise übertragen

MOVEQ #K,Dn — Übertrage »Quick«
L — #K(8 Bit) —> Dn

MULS ea,Dn — Multipliziere Signed
W — S*D —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*		*	*	*	*	*	*	*	*	*	*

MULU ea,Dn — Multipliziere Unsigned
W — S*D —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:*		*	*	*	*	*	*	*	*	*	*

NBCD ea — Negiere BCD-Zahl
B — 0-D-X —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

NEG ea — Negiere Operand
B W L — 0-D —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

NEGX ea — Negiere Operand mit X-Flag
B W L — 0-D-X —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

NOP — No Operation
tue nichts (dauert 4 Clock-Zyklen)

NOT ea
B W L

Logisch Nicht
-D —> D (Einer-Komplement)

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

OR ea,Dn / OR Dn,ea
B W L

Logisch ODER
S or D —> D

	DN	AN	(AN)	(AN)+	-(AN)	D(AN)	D(AN,RN)	$.W	$.L	D(PC)	D(PC,RN)	#
S:	*		*	*	*	*	*	*	*	*	*	*
D:	*		*	*	*	*	*	*	*			

ORI #K,ea
B W L

Logisch ODER mit Konstante
#K or D —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

ORI #K,CCR
B

Odere zu CCR
#K or CCR —> CCR

ORI #K,SR
W

Odere zu SR ! Privilegiert !
#K or SR —> SR

PEA ea
L

Push effektive Adresse
D —> -(SP)

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

RESET

Rücksetzen ! Privilegiert !
Reset-Leitung für 124 Clock-Zyklen auf 0

ROL Dn,Dn / ROL #K,Dn / ROL ea
B W L

Rotiere links
D n Bits rotiert —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:		W	W	W	W	W	W	W			

MS-Bit geht ins LS-Bit und ins Carry-Flag und schiebt links

ROR Dn,Dn / ROR #K,Dn / ROR ea — Rotiere rechts
B W L — D n Bits rotiert —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:		W	W	W	W	W	W	W			

LS-Bit geht ins MS-Bit und ins Carry-Flag und schiebt rechts

ROXL Dn,Dn / ROXL #K,Dn / ROXL ea — Rotiere links mit X-Flag
B W L — D n Bits rotiert —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:		W	W	W	W	W	W	W			

X geht ins LS-Bit und schiebt links. MS-Bit geht in X und Carry

---

ROXR Dn,Dn / RORL #K,Dn / RORL ea — Rotiere rechts mit X-Flag
B W L — D n Bits rotiert —> D

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:		W	W	W	W	W	W	W			

X geht ins MS-Bit und schiebt rechts. LS-Bit geht in X und Carry

---

RTE — Return von Exception — ! Privilegiert !
(Sp)+ —> SR; (SP)+ —> PC

RTR — Return mit Flag
(Sp)+ —> CCR; (SP)+ —> PC

RTS — Return
(SP)+ —> PC

SBCD Dn,Dn / ABCD -(An),-(An) — Subtrahiere BCD
B — D-S-X —> D

Scc — Setze Byte wenn cc true
B — if cc=true $FF —> Byte
else $00 —> Byte

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

STOP #K — Lade SR und Halt ! Privilegiert !
#K —> SR; Halt bis Interrupt

SUB ea,Dn / ADD Dn,ea — Subtrahiere
B W L — D-S —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	WL	*	*	*	*	*	*	*	*	*
D:	*		*	*	*	*	*	*	*			

SUBA ea,An — Subtrahiere Adresse
W L — D-S —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	*	*	*	*	*	*	*	*	*	*

Wortoperand wird wie bei EXT.L erweitert

SUBI #K,ea — Subtrahiere Konstante
B W L — D-#K —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

SUBQ #K,ea — Subtrahiere Konstante Quick (#K <=8)
B W L — D-#K —> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*	WL	*	*	*	*	*	*	*			

SUBX Dn,Dn / ADDX -(An),-(An) — Subtrahiere mit X-Flag
B W L — D-S-X —> D

SWAP Dn
W

Tausche Worte in Dn
Bit 31..16 <—> Bit 15..0

TAS ea
B

Teste und setze Bit 7 im Byte
Bit 7 —> N/Z-Flag
1 —> Bit 7

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

TRAP #n

Trap-Exception
PC —> -(SSP)
SR —> -(SSP)
Vektor n —> PC

TRAPV

Trap, wenn Overflow

TST ea
B W L

Teste Operand gegen Null
Ergebnis in N/Z-Flag

Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:*		*	*	*	*	*	*	*			

UNLK An

Unlink
An —> SP; (SP)+ —> An

Die Bedeutung der Condition Codes

	Kürzel	Bedeutung	Deutsch
	CC	Carry Clear	Carry = 0
	CS	Carry Set	Carry = 1
	EQ	Equal	Z =1
	GE	Greater or Equal	>=
***	GT	Greater Than	>
	HI	Higher	>
***	LE	Less or Equal	<
	LS	Less or Same	<=
***	LT	Less Than	<
	MI	Minus	-
	NE	Not Equal	<>
	PL	Plus	+
***	VC	oVerflow Clear	V = 0
***	VS	oVerflow Set	V =1
	T	True	1
	F	False	0

*** Für vorzeichenbehaftete Zahlen

Anhang A2
GEMDOS-Funktionen

Nr.	Name	Bedeutung und Aufruf	Rückgabe
$00	TERM	Rückkehr zum aufrufenden Programm clr -(sp) trap #1	
$01	CONIN	Zeichen von Tastatur lesen, darauf warten, Zeichen auf Schirm ausgeben (Echo) move #1,-(sp) trap #1 addq.l #2,sp	D0: High Word: Scan-Code Low Word : ASCII-Code

$02	CONOUT	Zeichen auf Schirm ausgeben
		move #Zeichen,-(sp)
		move #2,-(sp)
		trap #1
		addq.l #4,sp	

$03	AUXIN	Zeichen von RS-232 lesen, darauf warten move #3,-(sp) trap #1 addq.l #2,sp	D0: Low Word : Zeichen

$04	AUXOUT	Zeichen auf RS-232 ausgeben
		move #Zeichen,-(sp)
		move #4,-(sp)
		trap #1
		addq.l #4,sp	

$05	PRTOUT	Zeichen auf Drucker ausgeben
		move #Zeichen,-(sp)
		move #5,-(sp)
		trap #1
		addq.l #4,sp
			D0 = -1: OK
			= 0: Störung

Nr.	Name	Bedeutung und Aufruf	Rückgabe

$06	RAWIO	Fall a) CONIN ohne Warten
		move #$FF,-(sp)
		move #6,-(sp)
		trap #1
		addq.l #4,sp
			D0 = 0 wenn keine Taste gedrückt, sonst
			D0: High Word: Scan-Code
			Low Word : ASCII-Code

		Fall b) Erster Parameter <> $FF: Zeichenausgabe wie bei CONOUT	
$07	DIRCON	Zeichen von Tastatur lesen, darauf warten, kein Echo move #7,-(sp) trap #1 addq.l #2,sp	D0: High Word: Scan-Code Low Word : ASCII-Code
$08	DIRCON	Zeichen von Tastatur lesen, darauf warten, kein Echo (wirklich wie Nr. 7) move #8,-(sp) trap #1 addq.l #2,sp	D0: High Word: Scan-Code Low Word : ASCII-Code
$09	PRTLIN	Zeile (String) ausgeben pea Stringadresse move #9,-(sp) trap #1 addq.l #6,sp	
$0A	READLN	Zeile einlesen move.b #len,buffer pea buffer move #$0A,-(sp) trap #1 addq.l #6,sp	;Max. erlaubte Länge ;Pufferadresse D0: Ist-Länge buffer+0: Soll-Länge buffer+1: Ist-Länge buffer+2 f.: Text

Nr.	Name	Bedeutung und Aufruf	Rückgabe
$0B	CONSTAT	Status Console (genau: Tastaturpuffer) überprüfen move #$0B,-(sp) trap #1 addq-l #2,sp	D0 = -1: Zeichen im Puffer = 0: kein Zeichen

$0E	SETDRV	Drive zum aktuellen Drive ernennen
		move #drv,-(sp)
		move #$0E,-(sp)
		trap #1
		addq.l 4,sp
			;0=A, 1=B usw.
			D0: Nr. des vorherigen aktuellen Drives

$11	PRTSTAT	Status des Druckers an Parallel-Port move #$11,-(sp) trap #1 addq.l #2,sp	D0 = -1: Drucker bereit = 0: Drucker nicht bereit
$12	AUXINSTAT	Status des RS-232-Eingangs move #$12,-(sp) trap #1 addq.l #2,sp	D0 = -1: Zeichen verfügbar = 0: Kein Zeichen
$13	AUXOUTSTAT	Status des RS-232-Ausgangs move #$13,-(sp) trap #1 addq.l #2,sp	D0 = -1: Sender bereit = 0: Kein Zeichen senden
$19	CURDSK	Aktuelles Laufwerk ermitteln move #$19,-(sp) trap #1 addq.l #2,sp	D0: 0=A, 1=B usw.
$1A	SETDTA	Set Disk Transfer Address, DTA-Puffer einrichten pea dta_buffer move #$1A,-(sp) trap #1 addq.l #6,sp	;Adresse eines 44-Byte-Puffers

Nr.	Name	Bedeutung und Aufruf	Rückgabe

$20 SUPER In Supervisor Modus umschalten und zurück

a) in Supervisor-Mode schalten:

```
clr.l     -(sp)
move      #$20,-(sp)
trap      #1
addq.l    #6,sp                ;D0: SSP vor Aufruf
move.l    d0,ssp_save          ;sollte man retten
```

b) wieder in User-Modus:

```
move.l    ssp_save,-(sp)
move      #$20,-(sp)
trap      #1
addq.l    #6,sp
```

$2A GETDATE Datum lesen

```
move      #$2A,-(sp)
trap      #1
addq.l    #2,sp
```

DO: Bit 15 14 14 12 11 10 9 8 7 6 5 4 3 2 1 0
^..Jahr – 1980... ^ ^Monat ^ ^..Tag..^

$2B SETDATE Datum schreiben

```
move      #datum,-(sp)         ; Format wie $2A
move      #$2B,-(sp)
trap      #1
addq.l    #4,sp
```

$2C GETTIME Zeit lesen

```
move      #$2C,-(sp)
trap      #1
addq.l    #2,sp
```

DO: Bit 15 14 14 12 11 10 9 8 7 6 5 4 3 2 1 0
^Stunden.^ ^...Minuten.. ^ ^Sekunde ^ (durch 2)

$2D SETTIME Zeit schreiben

```
move      #zeit,-(sp)          ; Format wie $2D
move      #$2D,-(sp)
trap      #1
addq.l    #4,sp
```

Nr.	Name	Bedeutung und Aufruf		Rückgabe
$2F	GETDTA	Adresse des DTA-Puffers holen		
		move	#$2F,-(sp)	
		trap	#1	
		addq.l	#2,sp	D0: Adresse des DTA-Puffers
$30	VERSION	Lese TOS-Versions-Nummer		
		move	#$30,-(sp)	
		trap	#1	
		addq.l	#2,sp	D0: Versions-Nummer
$31	KEEP	Return zum Aufrufer, aber Speicher behalten		
		clr	-(sp)	
		move.l	#x,-(sp)	;x = Größe des Speichers
		move	#$31,-(sp)	
		trap	#1	
$36	FREDSK	Freien Bereich auf der Disk ermitteln		
		move	#1,-(sp)	;1=A, 2=B, 0=aktueller Drive
		pea	buffer	
		move	#$36,-(sp)	
		trap	#1	
		addq.l	#8,sp	
buffer		ds.l	1	;Anzahl freie Cluster
		ds.l	1	;Anzahl Cluster auf Disk
		ds.l	1	;Bytes pro Sektor (512)
		ds.l	1	;Sektoren pro Cluster (2)
$39	MAKEDIR	Sub-Directory (Ordner) anlegen		
		pea	name	;String mit Pfad-Namen
		move	#$39,-(sp)	
		trap	#1	
		addq.l	#6,sp	D0: Fehlernummer oder 0, wenn O.K.
$3A	RMDIR	Sub-Directory (Ordner) löschen		
		pea	name	;String mit Pfad-Namen
		move	#$3A,-(sp)	
		trap	#1	
		addq.l	#6,sp	D0: Fehlernummer oder 0, wenn O.K.

Nr.	Name	Bedeutung und Aufruf		Rückgabe
$3B	CHDIR	Sub-Directory wechseln		
		pea	name	;String mit Pfad-Namen
		move	#$3B,-(sp)	
		trap	#1	
		addq.l	#6,sp	D0: Fehlernummer oder 0, wenn O.K.
$3C	CREATE	Neues File öffnen		
		move	#0,-(sp)	;Status 0=read/write, 1=read
		pea	fname	;Filename
		move	#$3C,-(sp)	
		trap	#1	
		addq.l	#8,sp	D0: Handle-Nummer oder negative Fehler-Nr.
$3D	OPEN	Vorhandenes File öffnen		
		move	#0,-(sp)	;Status 0=read/write, 1=read
		pea	fname	;Filename
		move	#$3D,-(sp)	
		trap	#1	
		addq.l	#8,sp	D0: Handle-Nummer oder negative Fehler-Nr.
$3E	CLOSE	File schließen		
		move	#handle,-(sp)	;File-Handle
		move	#$3E,-(sp)	
		trap	#1	
		addq.l	#4,sp	D0: 0 oder negative Fehler-Nr.
$3F	READ	Anzahl Bytes aus File lesen		
		pea	buffer	;Adresse des Puffers
		move.l	#size,-(sp)	;Anzahl Bytes zu lesen
		move	#handle,-(sp)	;Handle
		move	#$3F,-(sp)	
		trap	#1	
		add.l	#12,sp	D0: Anzahl gelesene Bytes
$40	WRITE	Anzahl Bytes auf File schreiben		
		pea	buffer	;Adresse des Puffers
		move.l	#size,-(sp)	;Anzahl Bytes zu lesen
		move	#handle,-(sp)	;Handle
		move	#$4O,-(sp)	
		trap	#1	
		add.l	#12,sp	D0: Negativ: Fehler-Nummer

Nr.	Name	Bedeutung und Aufruf		Rückgabe
$41	DELETE	Löschen eines Files		
		pea	name	;(Pfad+)File-Name
		move	#$41,-(sp)	
		trap	#1	
		addq.l	#6,sp	D0: 0 = gelöscht <0 = Fehlernummer
$42	SEEK	File-Pointer (FP) stellen		
		move	#0,-(sp)	;0 = relativ zum Filebeginn ;1 = relativ zum Ist-FP ;2 = -relativ zu EOF
		move	#handle,-(sp)	;Handle aus OPEN
		move.l	#Bytes,-(sp)	;Versatz
		move	#$42,-(sp)	
		trap	#1	
		add.l	#10,sp	D0: <0 = Fehlernummer
$43	ATTRIB	File-Attribut lesen/setzen		
		move	#attrib,-(sp)	;0=rw, 1=r, 2=hidden, ;4=sys, 8=Disk-Name, 16=Dir
		move	#mode,-(sp)	;0=lesen, 1=setzen
		pea	name	;(Pfad+)File-Name
		move	#$43,-(sp)	
		trap	#1	
		add.l	#10,sp	D0: Bit-Vektor mit Attributen
$45	DUP	Standard-Handle duplizieren		
		move	#stdhandle,-(sp)	;0=Tastatur, 1=Schirm ;2=RS-232, 3=Drucker
		move	#$45,-(sp)	
		trap	#1	
		addq.l	#4,sp	D0: handle
$46	FORCE	Zwinge (leite um) Standard-Handle auf Handle		
		move	#handle,-(sp)	
		move	#stdhandle,-(sp)	;siehe $45
		move	#$46,-(sp)	
		trap	#1	
		addq.l	#6,sp	

Nr.	Name	Bedeutung und Aufruf			Rückgabe
$47	GETDIR	Aktuellen Directory-Pfad ermitteln			
		move	#drv,-(sp)		;Drive-Nummer
		pea	buffer		;für Ergebnis (>64 Bytes)
		move	#$47,-(sp)		
		trap	#1		
		addq.l	#8,sp		
$48	MALLOC	Speicher allokieren / freien Speicher ermitteln			
		move.l	#size,-(sp)		;>0=alloc., -1=ermitteln
		move	#$48,-(sp)		
		trap	#1		
		addq.l	#6,sp		D0: Startadresse des Speichers oder -1, wenn keiner mehr bzw. freie Bytes
$49	MFREE	Memory freigeben (was mit MALLOC allokiert war)			
		move.l	#adresse,-(sp)		;Adresse von MALLOC
		move	#$49,-(sp)		
		trap	#1		
		addq.l	#6,sp		D0: 0=OK, <0: Fehler
$4A	SETBLOCK	Speicherblock reservieren			
		move.l	#size,-(sp)		
		pea	adresse		
		clr	-(sp)		;Dummy
		move	#$4A,-(sp)		
		trap	#1		
		add.l	#12,sp		D0: 0=OK, sonst Fehler
$4B	EXEC	Programm nachladen / ausführen			
		pea	envi	;Adr. Environment String (0-Byte)	
		pea	cmd	;Adresse Kommando-Zeile (0-Byte)	
		pea	name	;Adresse File-Name	
		move	#mode,-(sp)	;0=lade u. starte, 1=lade	
		move	#$4B,-(sp)		
		trap	#1		
		add.l	#16,sp	D0:	>0: Adresse Base Page <0: Fehler
$4C	TERM	Programm beenden, Nachricht an Caller			
		move	#msg,-(sp)		;nur 1 Wort
		move	#$4C,-(sp)		
		trap	#1		

Nr.	Name	Bedeutung und Aufruf		Rückgabe
$4E	SFIRST	File suchen, erstes File in »*.*« suchen		
		move	#atr,-(sp)	;Attribut z. B. 0
		pea	name	;Suchstring
		move	#$4E,-(sp)	;
		trap	#1	
		addq.l	#8,sp	D0: <0 = nicht gefunden sonst Name in DTA
$4F	SNEXT	Nächstes suchen, Fortsetzung von $4E		
		move	#$4F,-(sp)	
		trap	#1	
		addq.l	#2,sp	D0: <0 = nicht gefunden sonst Name in DTA
$56	RENAME	File umbenennen		
		pea	neu	;Adresse neuer Name
		pea	alt	
		clr	-(sp)	;Dummy
		move	#$56.-(sp)	
		trap	#1	
		addl	#12,sp	D0 0=OK <0=Fehler
$57	GSDT	Get/Set Date+Time: Dateierstellungsdatum		
		move	#mode,-(sp)	;0=setzen, 1=lesen
		move	#handle,-(sp)	;Handle von OPEN
		pea	buffer	;Wort 1= Zeit, 2=Datum
		move	#$57,-(sp)	
		trap	#1	
		add.l	#10,sp	

Fehlermeldungen

-32 : Ungültige Funktionsnummer
-33 : File nicht gefunden
-34 : Pfad nicht gefunden
-35 : zu viele offene Dateien
-36 : Zugriff verboten (Attribut ändern)
-37 : ungültige Handle
-39 : nicht genug Speicher
-40 : ungültige Blockadresse
-46 : ungültiger Drive
-49 : Kein File mehr (SNEXT)

Anhang A3
BIOS-Funktionen

Nr.	Name	Bedeutung und Aufruf		Rückgabe
$00	GETMBP	Lade Memory-Parameter-Block		
		pea	block	;Adresse des MPB
		clr	-(sp)	
		trap	#13	
		addq.l	#6,sp	
$01	BCONSTAT	Status Eingabe-Geräte lesen		
		move	dev,-(sp)	;0=Drucker, 1=RS-232
				;2=Konsole, 3=MIDI
				;4=Tastatur-Prozessor
		move	#1,-(sp)	
		trap	#13	
		addq.l	#4,sp	D0: 0 = nicht bereit
				-1 = bereit
$02	BCONIN	Zeichen von Gerät lesen		
		move	dev,-(sp)	;dev wie bei $01
		move	#2,-(sp)	
		trap	#13	
		addq.l	#4,sp	D0: Zeichen
$03	BCONOUT	Zeichen an Gerät ausgeben		
		move	#zeichen,-(sp)	
		move	dev,-(sp)	;dev wie bei $01
		move	#3,-(sp)	
		trap	#13	
		addq.l	#6,sp	
$04	RWABS	Sektor lesen oder schreiben		
		move	#drv,-(sp)	;Drive: 0=A, 1=B usw.
		move	#recno,-(sp)	;relative Sektor-Nummer
		move	#secs,-(sp)	;Anzahl Sektoren
		pea	buffer	;Pufferadresse
		move	#flag,-(sp)	;0=Lesen, 1=Schreiben
		move	#4,-(sp)	
		trap	#13	
		add.l	#14,sp	D0: 0=O.K., <0=Fehler

Nr.	Name	Bedeutung und Aufruf		Rückgabe
$05	SETEXEP	Exception-Vektor setzen		
		pea	new_adress	;Adresse der neuen Routine
		move	#vec_no	;Nummer des Vektors
		move	#5,-(sp)	
		trap	#13	
		addq.l	#8,sp	D0: alte Adresse
$06	TICKCAL	Zeit zwischen 2 Aufrufen des Systemtimers ermitteln		
		move	#6,-(sp)	
		trap	#13	
		addq.l	#2,sp	D0: Zeit in ms
$07	GETBPB	Lese BIOS-Parameter-Block		
		move	drv,-(sp)	;Drive. 0=A, 1=B ...
		move	#7,-(sp)	
		trap	#13	
		addq.l	#4,sp	D0: Zeige auf BPB
$08	BCONSTAT	Status Ausgabe-Geräte lesen		
		move	dev,-(sp)	;0=Drucker, 1=RS-232
				;2=Konsole, 3=MIDI
				;4=Tastatur-Prozessor
		move	#8,-(sp)	
		trap	#13	
		addq.l	#4,sp	D0: 0 = nicht bereit
				-1 = bereit
$09	MEDIACH	Teste, ob Diskette gewechselt		
		move	drv,-(sp)	;0=A, 1=B ...
		move	#9,-(sp)	
		trap	#13	
		addq.l	#4,sp	D0: 0=nein, 1=jein, 2=ja
		Nicht empfehlenswert, funktioniert unsicher!!		
$0A	DRVMAP	Lese Drive Map		
		move	#$0A,-(sp)	
		trap	#13	
		addq.l	#2,sp	D0: Bit 0=A, Bit 1=B ...

Nr.	Name	Bedeutung und Aufruf			Rückgabe
$0B	KBSHIFT	Lesen/Setzen Status Sondertasten			
		move	#modus ,-(sp)		;-1=Lesen, >0 setzen
		move	#Bits,-(sp)	;Bit	0=Shift rechts
					1=Shift links
					2=Control
					3=Alt
					4=Caps ein
					5=Maus rechts
					6=Maus links
		trap	#13		
		add.l	#4,sp		D0: Bitvektor

Anhang A4
XBIOS-Funktionen

Nr.	Name	Bedeutung und Aufruf		Rückgabe
$00	INITMOUSE	Maus initialisieren		
		pea	adresse	;Adresse der Mausroutine
		pea	parms	;Zeiger auf Parameterblock
		move	#mode,-(sp)	;abs/relativ
		clr	-(sp)	
		trap	#14	
		add.	#12,sp	
$01	SSBRK	Speicher am oberen Speicherende reservieren		
		move	#bytes,-(sp)	
		move	#1,-(sp)	
		trap	#14	
		addq.l	#4,sp	
$02	PHYSBASE	Basisadresse des physikalischen VIDEO-RAM lesen		
		move	#2,-(sp)	
		trap	#14	
		addq.l	#2,sp	D0: Adresse
$03	LOGBASE	Basisadresse des logischen VIDEO-RAM lesen		
		move	#3,-(sp)	
		trap	#14	
		addq.l	#2,sp	D0: Adresse
$04	GETRES	Bildschirmauflösung lesen		
		move	#4,-(sp)	
		trap	#14	
		addq.l	#2,sp	D0: 0=niedrig, 1=mittel 2=hohe Auflösung
$05	SETSCREEN	Bildschirmparameter setzen (Wert oder -1=wie vor)		
		move	#-1,-(sp)	;Auflösung
		move.l	#-1,-(sp)	;phys. base
		move.l	#-1,-(sp)	;log. base
		move	#5,(sp)	
		trap	#14	
		add.l	#12,sp	

Nr.	Name	Bedeutung und Aufruf		Rückgabe
$06	PALETTE	Ganze Farbpalette laden		
		pea	pal_table	;Tabelle mit 16 Farbwerten
		move	#6,-(sp)	
		trap	#14	
		addq.l	#6,sp	
$07	COLOR	Einzelne Farbnummer neu laden		
		move	#farbe,-(sp)	; #0...#$777
		move	#no,-(sp)	;Farb-Nr. 0..15
		move	#7,-(sp)	
		trap	#14	
		addq.l	#6,sp	
$08	FLOPRD	Sektoren lesen		
		move	#count,-(sp)	;Anzahl Sektoren
		move	#side,-(sp)	;Seite (0,1)
		move	#track,-(sp)	;Spur
		move	#sec,-(sp)	;Sektor (1..9 (10))
		move	#drv,-(sp)	;Drive 0=A, 1=B ...
		clr.l	-(sp)	;Dummy
		pea	buffer	;Pufferadresse
		move	#8,-(sp)	
		trap	#14	
		add.l	#20,sp	D0: 0=OK, <0=Fehler
$09	FLOPWR	Sektoren schreiben		
		move	#count,-(sp)	;Anzahl Sektoren
		move	#side,-(sp)	;Seite (0,1)
		move	#track,-(sp)	;Spur
		move	#sec,-(sp)	;Sektor (1..9 (10))
		move	#drv,-(sp)	;Drive 0=A, 1=B ...
		clr.l	-(sp)	;Dummy
		pea	buffer	;Pufferadresse
		move	#9,-(sp)	
		trap	#14	
		add.l	#20,sp	D0: 0=OK, <0=Fehler
$0A	FLOPFMT	Diskette formatieren		
		move	#$E5E5,-(sp)	;Testmuster
		move.l	#$87654321,-(sp)	;Ataris Magic
		move	#1,-(sp)	;Interleave
		move	#side,-(sp)	;Seite (0,1)
		move	#track,-(sp)	;Spur
		move	#sec,-(sp)	;Sektor (1..9 (10))
		move	#drv,-(sp)	;Drive 0=A, 1=B ...

Nr.	Name	Bedeutung und Aufruf		Rückgabe
		clr.l	-(sp)	;Dummy
		pea	buffer	;Adresse Puffer $2000 Byte
		move	#$0A,-(sp)	
		trap	#14	
		add.l	#26,sp	D0: 0=OK, <0=Fehler
$0C	MIDIWS	String auf MIDI ausgeben		
		pea	string	
		move	#len, (-sp)	;Länge des String
		move	#$0C,-(sp)	
		trap	#14	
		addq.l	#8,sp	
$0D	MFPINIT	MFP-Interrupt initialisieren		
		pea	adresse	;Adr. Interrupt-Handler
		move	#no,-(sp)	;Vektor-Nr. (0..15)
		move	#$0D,-(sp)	
		trap	#14	
		addq.l	#8,sp	
$0E	IOREC	Adresse IO-Record eines Gerätes lesen		
		move	#dev,-(sp)	;0=RS, 1=Tastat., 2=MIDI
		move	#$0E,-(sp)	
		trap	#14	
		addq.l	#4,sp	D0: Zeiger auf IO-Rec
				L: Zeiger auf Ringpuffer
				W: Größe Ringpuffer
				W: Headpointer
				W: Tailpointer
				W: Startindex (XON/RTS)
				W: Stopindex (XOFF/CTS
$0F	RSCONF	RS-232 konfigurieren (Wert oder -1=wie vor)		
		move	#-1,-(sp)	;scr im MFP
		move	#-1,-(sp)	;tsr im MFP
		move	#-1,-(sp)	;rsr im MFP
		move	#-1,-(sp)	;ucr im MFP
		move	#-1,-(sp)	;1=XON/XOFF, 2=RTS/CTS
		move	#-1,-(sp)	;0..9=19200..300 Baud
		move	#$0F,-(sp)	
		trap	#14	
		add.l	#14,sp	

Nr.	Name	Bedeutung und Aufruf		Rückgabe
$10	KEYTBL	Tastaturbelegung ändern		(Wert oder -1=wie vor)
		pea	caps	; CAPS-Tabelle
		pea	shift	; Shift-Tabelle
		pea	unshift	; Unshift-Tabelle
		move	#$10,-(sp) ;	je 128 Bytes
		trap	#14	
		add.l	#14,sp	D0: Zeiger auf den ersten der drei Vektoren
$11	RANDOM	Lesen ein 24-Bit-Zufallszahl		
		move	#$11,-(sp)	
		trap	#14	
		addq.l	#2,sp	D0: Zufallszahl
$12	BOOTSEC	Boot-Sektor erzeugen		(Wert oder -1=wie vor)
		move	#exec,-(sp)	; 0=nicht, 1=ausführbar
		move	#typ,-(sp)	; 2=ein-, 3=zweiseitig
		nove.l	#no,-(sp)	; 24-Bit Seriennummer
		pea	buffer	; 512 Bytes Boot-Sektor
		move	#$12,-(sp)	
		trap	#14	
		add.l	#14,sp	
$13	FLOPVER	Floppy-Sektoren verifizieren		
		move	#count,-(sp)	;Anzahl Sektoren
		move	#side,-(sp)	;Seite (0,1)
		move	#track,-(sp)	;Spur
		move	#sec,-(sp)	;Sektor (1..9 (10))
		move	#drv,-(sp)	;Drive 0=A, 1=B ...
		clr.l	-(sp)	;Dummy
		pea	buffer	;Sektoren im RAM
		move	#$13,-(sp)	
		trap	#14	
		add.l	#20,sp	D0: 0=OK, <0=Fehler
$14	SCRNDMP	Schirm auf Drucker kopieren (wie Alt-Help)		
		move	#$14,-(sp)	
		trap	#14	
		addq.l	#2,sp	

Nr.	Name	Bedeutung und Aufruf	Rückgabe
$15	CURSOR	Cursor an/aus und Blinkfrequenz setzen	
		move #freq,-(sp)	;frq/70 sec
		move #n,-(sp)	;0: aus, 1=an, 2=blinkend
			;3=fest, 4=setze Frequenz
			;5=lese
		move #$15,-(sp)	
		trap #14	
		addq.l #6,sp	D0: (alte) Frequenz
$16	SETTIME	Datum und Zeit setzen	
		move.l #dt,-(sp)	
		move #$16,-(sp)	
		trap #14	
		addq.l #6,sp	

dt: Bit 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
^..Jahr – 1980....^ ^..Monat.^ ^.... Tag ....^

dt: Bit 15 14 14 12 11 10 9 8 7 6 5 4 3 2 1 0
^Stunden.^ ^.....Minuten .... ^ ^..Sekunden.^ (durch 2)

$17	GETTIME	Datum und Zeit lesen	
		move #$17,-(sp)	
		trap #14	
		addq.l #2,sp	D0: Datum/Zeit wie $16
$18	BIOSKEYS	Ursprüngliche Tastaturbelegung (wieder)herstellen	
		move #$18,-(sp)	
		trap #14	
		addq.l #2,sp	
$19	IKBWS	Strings (Befehle) an IKB senden	
		pea string,-(sp)	
		move #len,-(sp)	;Länge
		move #$19,-(sp)	
		trap #14	
		addq.l #8,sp	
$1A	DISINT	Disable Interrupt des MFP	
		move #no,-(sp)	;Nr. (0..15)
		move #$1A,-(sp)	
		trap #14	
		addq.l #4,sp	

Nr.	Name	Bedeutung und Aufruf		Rückgabe
$1B	DISINT	Enable Interrupt des MFP		
		move	#no,-(sp)	;Nr. (0..15)
		move	#$1B,-(sp)	
		trap	#14	
		addq.l	#4,sp	
$1C	GIACCESS	Daten an Sound-Chip senden		
		move	#reg,-(sp)	;Register-Nr.
		move	#data,-(sp)	;Daten
		move	#$1C,-(sp)	
		trap	14	
		addq.l	#6,sp	
$1D/$1E	Bit im »Sound«-Chip/Port A setzen/löschen			
		move	#Bit,-(sp)	;siehe unten
		move	#$1D/$1E,-(sp)	;setzen/löschen
		trap	#14	
		addq.l	#4,sp	

Bit	Wirkung
0	Floppy-Seite 0/1
1	Drive A
2	Drive B
3	RS-232 RTS
4	DTS
5	Centronic Strobe

$1F	TIMER	Timer des MFP einer Interr.-Routine zuweisen + starten		
		pea	adr	;Adresse der Interrupt-Routine
		move	#data,-(sp)	;—> Daten-Register
		move	#ctrl,-(sp)	;—> Control-Register
		move	#timer,-(sp)	;0..3=A..D
		move	#$1F,-(sp)	
		trap	#14	
		add.l	#12,sp	
$20	DOSOUND	Ton erzeugen		
		pea	string	;Sound-Befehle
		move	#$20,-(sp)	
		trap	#14	
		addq.l	#6,sp	

Nr.	Name	Bedeutung und Aufruf	Rückgabe
$21	SETPRT	Drucker-Konfiguration setzen/lesen	(Wert oder -1)

```
move    #bits          ;siehe unten
move    #$21,-(sp)
trap    #14
addq.l  #4,sp          D0: Bits
```

Bit	0	1
0	Matrix	Typenrad
1	s/w	Farbe
2	IBM	Epson
3	Test	Qualität
4	Parallel	RS-232
5	Endlos	Einzelblatt

$23 KBREP Tastatur-Repeat ändern oder lesen(Wert=-1)

```
move    #freq,-(sp)
move    #delay,-(sp)
move    #$23,-(sp)
trap    #14
addq.l  #6,sp          D0.W Hi: delay, Low: freq
```

$25 WVBL Wait for Vertikal Blank (Austastlücke)

```
move    #$25,-(sp)
trap    #14
addq.l  #2,sp
```

$26 SUPEREXEC Unterprogramm im Supervisor-Modus ausführen

```
pea     adr            ;Adresse des UP's
move    #$26,-(sp)
trap    #14
addq.l  #6,sp
```

$27 PUNTAES AES löschen, wenn nicht im ROM

```
move    #$27,-(sp)
trap    #1
addq.l  #2,sp
```

Anhang A5
Line-A-Grafik

Offsets in Tabelle der Line-A-Variablen

```
v_planes  equ   0    ;Anzahl Video-Planes
v_lin_wr  equ   2    ;Bytes/Video-Zeile
contrl    equ   4    ;Zeiger auf  Contrl-Array
intin     equ   8    ;            Intin-Array
ptsin     equ   12   ;            Points-In-Array
intout    equ   16   ;            Intout-Array
ptsout    equ   20   ;            Points-Out-Array
fgbp1     equ   24   ;Die 4 Bit-Planes
fgbp2     equ   26   ;  fuer
fgbp3     equ   28   ;    die
fgbp4     equ   30   ;      Farbe
lstlin    equ   32   ;sollte -1 sein
ln_mask   equ   34   ;Linienmuster
wrt_mode  equ   36   ;Schreibmodus
x1        equ   38   ;Ausgangspunkt
y1        equ   40
x2        equ   42   ;Endpunkt
y2        equ   44
patptr    equ   46   ;Zeiger —> Fuellmuster
patmsk    equ   50   ;Maske Fuellmuster
multifil  equ   52   ;0/1: 1/alle Planes fuellen
clip      equ   54   ;0/1: Clipping aus/an
xmin_clp  equ   56   ;Clipping-Rechteck
ymin_clp  equ   58   ;
xmax_clp  equ   60   ;
ymax_clp  equ   62   ;
```

Die Grafik-Routinen

$A000 Initialisierung

Eingang: ---
Ausgang: D0,A0 : Zeiger auf Line-A-Variable
A1 : Zeiger auf Tabelle mit Startadressen der drei System-Fonts
A2 : Zeiger auf Sprungtabelle der Line-A-UP's

$A001 Put Pixel (Bildpunkt setzen)

Eingang: PTSIN(0) = x
PTSIN(1) = y
INTIN(0) = Farbe

$A002 Get Pixel (Bildpunkt lesen)

Eingang: PTSIN(0) = x
PTSIN(1) = y
Ausgang: D0 = Farbe

$A003 Line Linie zeichnen

Eingang:
x1, y1 = »von« x,y
x2, y2 = »bis« x,y
fgbp1 = Plane 1 (alle)
fgbp2 = Plane 2 (640x200, 320x200)
fgbp3 = Plane 3 (nur 320x200)
fgbp4 = Plane 4 (nur 320x200)
ln_mask = Linienmuster ($FFFF = durchgehend)
wrt_mod = 0..3 = replace, transparent, xor, inverse
lstlin = -1

$A004 Horizontale Linie

Eingang:
x1, y1 = von x,y
x2 = bis x
fg_bb_1 = Plane 1 (alle)
fgbp2 = Plane 2 (640x200, 320x200)
fgbp3 = Plane 3 (nur 320x200)
fgbp4 = Plane 4 (nur 320x200)
ln_mask = Linienmuster ($FFFF = durchgehend)
wrt_mod = 0..3 = replace, transparent, xor, invers
lstlin = -1
pat_ptr = Zeiger auf Liste Worte mit Linienmustern
Ab _pat_ptr + _pat_mask müssen > _pat_mask Muster stehen
pat_mask = siehe _pat_ptr

$A005 Gefülltes Rechteck

Eingang:
x1, y1 = linke obere Ecke
x2, y2 = rechte untere Ecke
Clip = Clipping Flag
wenn Clipping (Clipping Flag=1):
xmin_clp,
xmax_clp,
ymin_clp,
ymax_clp = Clipping-Rechteck
fgbb1 = Plane 1 (alle)
fgbp2 = Plane 2 (640x200, 320x200)
fgbp3 = Plane 3 (nur 320x200)
fgbp4 = Plane 4 (nur 320x200)
wrt_mod = 0..3 = replace, transparent, xor, invers
lstlin = -1
pat_ptr = Zeiger auf Liste Worte mit Linienmustern
Ab _pat_ptr + _pat_mask müssen > _pat_mask Muster stehen
pat_mask = siehe _pat_ptr

$A006 Gefülltes Polygon

Eingang:
ptsin = Array mit xy-Koordinaten je Eckpunkt
contrl[1] = Anzahl der Eckpunkte
y1 = Y-Wert der zu füllenden Zeile (ggf. mehrfach aufrufen)
wenn Clipping (Clipping Flag=1):
xmin_clp,
xmax_clp,
ymin_clp,
ymax_clp = Clipping-Rechteck
fgbb1 = Plane 1 (alle)
fgbp2 = Plane 2 (640x200, 320x200)
fgbp3 = Plane 3 (nur 320x200)
fgbp4 = Plane 4 (nur 320x200)
wrt_mod = 0..3 = replace, transparent, xor, invers
lstlin = -1
pat_ptr = Zeiger auf Liste Worte mit Linienmustern
Ab _pat_ptr + _pat_mask müssen > _pat_mask Muster stehen
pat_mask = siehe _pat_ptr

$A009 Mauszeiger an

$A00A Mauszeiger aus

$A00B Maus-Form

Eingang:
intin[3] = 0
intin[4] = 1
intin[5] .. intin[20] = 16 Worte der Maske
intin[21] .. intin[36] = 16 Worte Cursor-Daten

$A00C Sprite ausschalten

Eingang:
A2 = Zeiger auf Sicherungsbereich (siehe $A00D

$A00D Sprite zeichnen

Eingang:
D = X-Koordinate
D1 = und Y
A0 = Zeiger auf Sprite-Daten
A2 = Zeiger auf Sicherungsbereich

Sprite-Daten

```
        dc.w    8           ;X-Offset vom Hot Spot
        dc.w    8           ;Y
        dc.w    -1          ;XOR-Sprite-Format
        dc.w    0           ;Hintergrundfarbe (keine)
        dc.w    1           ;Vordergrund schwarz

        dc.w    %0000000000000000   ;Hintergrund    Zeile 0
        dc.w    %0000001111000000   ;Vordergrund    Zeile 0
        dc.w    %0000000000000000   ;Hintergrund    1
        dc.w    %0000111111110000   ;Vordergrund    1
        dc.w    %0000001111000000   ;usw. bis Zeile 15

sprsave ds.b    74          ;Speicher für Sprite (266 Bytes in Farbe)
```

Anhang A6
Der Zeichensatz des Atari-ST

Die folgende Tabelle bringt alle auf dem ST-Bildschirm darstellbaren Zeichen. Zur Ermittlung des Codes eines Zeichens lesen Sie zuerst in der Spalte ganz links das erste Hex-Digit, dann über dem Zeichen das zweite. Zum Beispiel hat A den Code 41. Natürlich wurde auch dieses Bild in Assembler erzeugt. Das Listing dazu finden Sie auf der folgenden Seite.

Bild A.6.1 Der Zeichensatz des Atari-ST

*** CHARS Zeichensatz des ST ausgeben**

```
conin     macro
          move.w   #1,-(sp)        ;Funktion CONIN
          trap     #1              ;aufrufen
          addq.l   #2,sp           ;Stack korrigieren
          endm

writes    macro
          pea      \1
          move.w   #9,-(sp)        ;Funktion PRINTLINE
          trap     #1
          addq.l   #6,sp
          endm

bconout   macro
          move     \1,-(sp)        ;Zeichen
          move     \2,-(sp)        ;Device
          move     #3,-(sp)        ;Funktion bCONOUT
          trap     #13
          addq.l   #6,sp
          endm

          include  "linea.inc"     ;siehe Kapitel 9

          writes   hex1            ;A-F waagerecht
          writes   crlf
          lea      hex2,a4         ;A-F senkrecht
          move     #0,d4           ;auszugebendes Zeichen
          move     #15,d5          ;Schleifenzaehler
a_loop    move     #15,d6          ;eine Zeile a 16 Zeichen
          move.b   (a4)+,d7        ;A..F
          bconout  d7,#5           ;ausgeben
          bconout  #32,#5          ;Blank hinterher
i_loop    bconout  d4,#5           ;Zeichen ausgeben
          bconout  #32,#5          ;noch ein Blank
          addq     #1,d4           ;naechstes Zeichen
          dbra     d6,i_loop       ;Schleife 1 Zeile
          writes   crlf            ;neue Zeile
          dbra     d5,a_loop       ;bis alle 16
```

Nun die Linien zeichnen

```
        init    ;Line-A initialisieren
        color   #1,#1,#1,#1         ;Farbe s/w
        move    #0,wrt_mode(a5)     ;Relace-Mode
        move    #-1,ln_mask(a5)     ;durchgezogene Linie
        move    #16,d3              ;Schleifenzaehler
        move    #12,d4              ;Start Linien
lin     line    d4,#0,d4,#272       ;senkrechte Linien
        add     #4,d4               ;waagerechte etwas tiefer
        line    #0,d4,#268,d4       ;nun zeichnen
        add     #12,d4              ;naechste Linien
        dbra    d3,lin              ;

        conin                       ;auf Taste warten
        clr     -(sp)               ;Terminate
        trap    #1

crlf    dc.b    13,10,0             ;Zeilenvorschub
        ds.w    0
hex1    dc.b    '  0 1 2 3 4 5 6 7 8 9 A B C D E F',0
        ds.w    0
hex2    dc.b    '0123456789ABCDEF'

        end
```

Bild A.6.2 Ausgabe des Zeichensatzes

Das Programm »lebt« von dem wenig bekannten und meines Wissens auch nicht offiziell dokumentierten Gerät 5. Wird Gerät 5 als »Device« in der Funktion BCONOUT angezogen, werden die Zeichen 0..31 nicht mehr als Control-Codes interpretiert, sondern dann werden die hübschen Zeichen laut Bild A.6.1 generiert.
Die eigentliche Zeichenausgabe beginnt bei »a_loop«. Die 256 möglichen Zeichen werden in zwei geschachtelten Schleifen, (jede bis 16) im Register D4 hochgezählt. Die Technik der zwei Schleifen hat den Vorteil, daß man sehr einfach den Zeilenvorschub nach 16 Zeichen realisieren kann. Der zweite Teil des Listings dient nur dem Zeichnen des Gittermusters mittels der Line-A-Grafik, wie wir sie schon aus Kapitel 9 kennen.

Anhang A7
Der VT52-Emulator

VT52 heißt Video Terminal, Modell 52.

Es wurde früher lange Zeit an DEC-Computern eingesetzt, heute allerdings kaum noch. Geblieben ist der Standard, der beschreibt, welche Steuerzeichen welche Reaktion auf dem Bildschirm auslösen.

Wenn man diese Steuersequenzen an den Bildschirm des Atari ST sendet, verhält er sich wie ein VT52. In Assembler können Sie dafür die GEMDOS/BIOS-Funktionen CONOUT/BCONOUT (Konsolausgabe) einsetzen.

Jede Steuersequenz beginnt mit dem Zeichen »Escape« ($1B, dezimal 27). Dem »Escape« folgt dann mindestens ein weiteres Zeichen, das im ASCII-Code >32 ist.

In der folgenden Liste ist »Escape« wie üblich mit »Esc« abgekürzt.

Esc A — Cursor up (aufwärts)
Der Cursor wird eine Zeile nach oben bewegt. Ist er schon in der ersten Zeile, erfolgt keine Reaktion

Esc B — Cursor down (abwärts)
Der Cursor wird eine Zeile nach unten bewegt. Ist er schon in der letzten Zeile, erfolgt keine Reaktion

Esc C — Cursor right (nach rechts)
Der Cursor wird um eine Spalte nach rechts bewegt. Ist er schon ganz rechts, erfolgt keine Reaktion

Esc D — Cursor left (nach links)
Der Cursor wird um eine Spalte nach links bewegt. Ist er schon ganz links, erfolgt keine Reaktion

Esc E — Clear home
Der Schirm wird gelöscht, der Cursor geht nach links oben.

Esc H — Cursor home
Der Cursor geht nach links oben

Esc I — Cursor up and insert
Der Cursor geht um eine Zeile nach oben. Ist er schon in der ersten Zeile, wird eine Leerzeile eingefügt. Die Folgezeilen rücken dann nach unten

Esc J — Clear to end of frame
Der Bildschirm wird ab Cursor bis zum Ende gelöscht

Esc K — Clear to end of line
Die Zeile wird ab Cursor gelöscht

Esc L — Insert line
Leerzeile wird an aktueller Cursor-Zeile eingefügt. Der Rest des Schirms wird um eine Zeile nach unten geschoben

Esc M — Delete Line
Löscht Zeile, in der sich der Cursor befindet. Die anderen Zeilen rücken auf. Die letzte Zeile wird leer

Esc Yxy — Move Cursor
Bewegt den Cursor auf Spalte x, Zeile y. Da nur ASCII-Codes >32 als Argument zugelassen sind, muß auf die X- und Y-Position immer 32 addiert werden. Diese Summen sind dann die »Zeichen«, die nach Esc Y noch zu senden sind.

Bereiche:	Spalten (X)	= 0..79	=	Code 32..111
	Zeilen (Y)	= 0..24	=	Code 32.. 56

Achtung: Ab jetzt Kleinbuchstaben!

Esc b — Vordergrund-Farbe (Schriftfarbe)
Ausgewertet werden nur die vier niederwertigen Bits des folgenden Zeichens. Daher ist es egal, ob man für die Farbe 1 das Zeichen 1 oder den Buchstaben A oder a nimmt. Null oder @ ist weiß. Auf monochromen Monitoren ist nur 0 oder 1 (schwarz) möglich. Bei mittlerer Auflösung sind 4, in niedriger Auflösung 16 Farben möglich

Esc c — Hintergrund-Farbe
Es gel n die Regeln von Esc_b

Esc d — Clear frame to Cursor
Der Schirm wird vom Anfang bis zum Cursor gelöscht

Esc e — Cursor on
Cursor einschalten

Esc f — Cursor off
Cursor ausschalten

Esc j — Store Cursor
Die aktuelle Cursor-Position wird gespeichert

Esc k — Restore Cursor
Der Cursor wird auf die mit Esc_j gespeicherte Position bewegt

Esc l — Clear line
Die Cursor-Zeile wird gelöscht. Der Cursor geht dann auf den Zeilenanfang

Esc o — Clear line to Cursor
Die Cursor-Zeile wird vom Anfang bis zum Cursor gelöscht

Esc p — Inverse on
Vordergrund- und Hintergrundfarbe werden getauscht
In s/w wird schwarz auf weiß geschrieben

Esc q — Inverse off
Hebt Esc_p wieder auf

Esc v — Wrap on
In diesem Modus wird bei Texten, die über die Spalte 79 hinausgehen, eine neue Zeile begonnen

Esc w — Wrap off
Hebt Esc_v wieder auf. Alle Zeichen nach Spalte 79 werden in Spalte 79 geschrieben

Anhang A8
Die Systemvariablen

Im folgenden werden genannt:

Der Name lt. Atari-Dokumentation
Der Typ (B, W, L)
Die Adresse
Eine kurze Erklärung

evt_timer	L	$400	:	Zähler für Timer des GEM
evt_critic	L	$404	:	Zeiger auf Error-Handler
evt_term	L	$408	:	Zeiger auf Terminate-Routine
evt_xtra	L	$40C	:	Reserve für 5 weitere Vektoren
memvalid	L	$420	:	$752019F3, wenn Kaltstart O.K.
memctrl	B	$424	:	4=512 K, 5=1024 K RAM
resvalid	L	$426	:	Wenn $31415926, wird bei Reset über den Vektor in $42A gesprungen
resvector	L	$42A	:	siehe $426
phystop	L	$42E	:	Zeiger auf RAM-Ende + 1
_membot	L	$432	:	Beginn Programmspeicher (TPA)
_memtop	L	$436	:	Ende der TPA
memval2	L	$43A	:	$237698AA, wenn Kaltstart O.K.
flock	W	$43E	:	Wenn >0 DMA gesperrt, weil VBL
seekrate	W	$440	:	00=6, 01=12, 10=2, 11=3 ms (Ist-Stand)
_timr_ms	W	$442	:	Timer-Intervall (20 ms)
_fverify	W	$444	:	Verify-Flag. Wenn <>0, erfolgt ein Lesen nach jedem Floppy-Schreiben

_bootdev	W	$446	:	Nummer des Boot-Drives (0=A)
palmode	W	$448	:	0 = NTSC, 1 = PAL-Video
defshiftmd	B	$44A	:	Auflösung. 0=low, 1=medium, 2=high
_v_base_ad	L	$44E	:	Zeiger auf Video-RAM
vblsem	W	$452	:	Wenn 1, VBL-Interrupts erlaubt
nvbls	W	$454	:	Anzahl der VBL-Routinen (8)
_vblqueue	L	$456	:	Zeiger auf Liste der VBL-Handler
colorptr	L	$45A	:	Zeiger auf 16 Farbworte der Palette Wenn 0, wird Farbe nicht geändert
screept	L	$45E	:	Wenn >0, neue Video-Basis
_vbclock	L	$462	:	Zähler der VBL-Interrupts
_frclock	L	$466	:	Anzahl der ausgeführten VBL-Interrupts (nicht durch VBL-Semaphor gesperrt)
hdv_init	L	$46A	:	0 oder Zeiger zur Hard-Disk-Init.
reserved	L	$46E	:	Reserviert für spätere Verwendung
hdv_bpb	L	$472	:	Hard-Disk-Link für BIOS-Parameter-Block
hdv_rw	L	$476	:	Hard-Disk-Link für Read/Write-Routine
hdv_boot	L	$47A	:	Zeiger auf Boot-Routine von Hard-Disk
hdv_mediach	L	$47E	:	Hard-Disk-Link für Media Change
_cmdload	W	$482	:	Wenn <>0, COMMAND.PRG anstatt Desktop laden
conterm	B	$484	:	Bit 0 = ^G gibt Ton ein/aus 1 = Tastatur-Repeat ein/aus 2 = Tastatur-Klick ein/aus
themd	L	$48E	:	Memory-Descriptor. Wird von BIOS getmbb überschrieben
savptr	L	$4A2	:	Sicherungsbereich Trap-Dispatcher (siehe Kapitel 15)

nflops	L	$4A6	:	Anzahl der angeschlossenen Floppies
con_state	L	$4A8	:	Vektor für Bildschirmausgabe (VT52)
save_row	W	$4AC	:	Cursor-Zeile
save_context	L	$4AE	:	Sicherungsbereich Prozessor-Status bei Exceptions
_bufl	L	$4B2	:	Zeiger auf 2 BCB
_Hz_200	L	$4BA	:	Zähler für 200 Hz Systemtakt
the_eenv		$4BC	:	Default Environment String ('0000')
_drvbits	L	$4C2	:	Je angemeldetem Drive 1 Bit = 1
_dskbufp	L	$4C6	:	Zeiger auf 1 K Puffer für Disk-I/O
_prt_cnt	W	$4EE	:	-1, wenn >0, Screendump (Alt-Help)
_prt_abt	W	$4F0	:	Printer-Abort-Flag, wenn Timeout
_sysbasse	L	$4F2	:	Zeiger auf Beginn des TOS
_sys_end	L	$4FA	:	Zeiger auf Ende des TOS

Anhang A9
Liste der Exception-Vektoren des ST

Vektor	Inhalt
0	Stackpointer nach Reset
1	PC nach Reset
2	Bus-Error
3	Adreß-Error
4	Illegaler Befehl
5	Division durch Null
6	CHK-Test
7	TRAPV
8	Privileg-Verletzung
9	Trace
10	Line-A-Emulator (Grafik-Kernroutinen)
11	Line-F-Emulator
12-14	reserviert
15	Nicht initialisierter Interrupt
16-23	reserviert
24	falscher Interrupt
25	Interrupt Autovektor Ebene 1
26	Interrupt Autovektor Ebene 2
27	Interrupt Autovektor Ebene 3
28	Interrupt Autovektor Ebene 4
29	Interrupt Autovektor Ebene 5

Vektor	Inhalt	
30	Interrupt Autovektor Ebene 6	
31	Interrupt Autovektor Ebene 7	
32	Trap #0	
33	Trap #1	GEMDOS
34	Trap #2	GEM
35	Trap #3	
36	Trap #4	
37	Trap #5	
38	Trap #6	
39	Trap #7	
40	Trap #8	
41	Trap #9	
42	Trap #10	
43	Trap #11	
44	Trap #12	
45	Trap #13	BIOS
46	Trap #14	XBIOS
47	Trap #15	
48-63	reserviert	
64-79	beim ST Interrupt-Vektoren des MFP 68901	
80-255	Weitere Anwender-Interrupt-Vektoren	

Mit Ausnahme der Division durch Null wird bei Zugriff auf die Fehler-Vektoren oder nicht belegte Vektoren eine Routine aufgerufen, die die Vektor-Nummer als Anzahl »Bomben« ausgibt und dann das Programm abbricht.
Die reservierten Vektoren sind von Motorola reserviert und sollten von Ihnen nicht verwendet werden. Die freien Traps stehen Ihnen zur Verfügung, dto. die Nonauto-Vektoren 80–225.

Stichwortverzeichnis

2er-Komplement-Zahlen 87
40-Zeichen-Bildschirm 139
40-Zeichen-Videocontroller 139, 196
40/80-Zeichen-Taste 137

A-Befehl 40
A11 112
AAX 112
Abbruch
-, der Assemblierung 207
-, des Listens 182
-, des Testlaufes 207
Abkürzung 179
Abspeichern 162, 165, 185, 205, 316
ADC 94
Addition 190
Addition, binäre Addition 70
Adresse 170
Adressenregister 21
Adressierung 42, 171, 309, 315
-, absolut-indizierte Adressierung 45, 46
-, absolute Adressierung 43, 310
-, implizite Adressierung 42
-, indirekt-absolute Adressierung 45
-, indirekt-indizierte Adressierung 48
-, indiziert-indirekte Adressierung 47
-, relative Adressierung 44
-, relative-Test-Bit-Adressierung 49
-, Set/Reset-Bit-Adressierung 49
-, unmittelbare Adressierung 43
-, Zeropage-Adressierung 44
-, zeropage-indizierte Adressierung 46
Adressierungsart 42, 171, 171
Adressanpassung 310
Adressbereich 306
Adressblock 328
Adressbus 22, 29
Adressleitung 22
Adressraum 128
Adresstabelle, WORD 218
Adresswert 280
Akkumulator 23
Akkumulator-Adressierung 43
.alter 254
ALU 23
AND 96
.append 226
Append-Verkettung 226
Arbeitsbereich 182
Arbeitsdiskette 168
ARR 112
ASCII 21, 272, 307
-, Dumps 307
-, Tabelle 170
-, Texte 215
-, Wert 190
-, DIN-Schalter 135
ASE 168
ASE-Befehle 351
ASE-Editor 175
ASE-Linker 281
ASE-Token 256
ASL 98
ASR 112
Assembler 15, 16, 171
-, Direktassembler 171
-, Line Assembler 171
-, Zeilenassembler 171
Assemblerlisting 181, 269
Assemblerpass 274
Assemblierung 39, 174, 197, 213, 254, 259, 272, 278
-, Startadresse der Assemblierung 197, 201
Aufheben des Befehles NEW 195
Ausdruck 184, 202
Ausgabe, eines Textes 155
-, eines Zeichens 155
-, einer Datei 162
Aussage 76
auto 180
AXS 112

Bank 22
Banking 142
Banknummer 142
Bankswitching 128
.base 202
Basic-Interpreter 129
Basic-ROM 129, 134
Basicvariable 164
Basin 160, 164
BBR 119
BBS 119
BCC 84
BCD-Zahl 59, 75
BCS 84
Befehlserweiterung 166
Befehlsregister 27
Befehlsverteiler 217
.begin 210
BEQ 85
Betriebssystem 130
Bildkode
-, Tabelle 170
-, Text 215, 221
-, Wert 190
Bildschirm 305
-, Editor 304
-, Fenster 155, 156, 304
Binärsystem 18
Binärzahl 72, 189
Bit 18, 61, 106, 107, 170
Bit-Befehl 78
Bit-Maske 79
Bitschiebebefehl 98
Blank 93
Block 210
-, Anzahl 211
-, Vergleich 311
-, Verschiebung 238, 309
BMI 86
BNE 85
Boolesche Algebra 18, 76
boot 198
Boot-Sektor 313
Boot-Vorgang 197
Bootstrap 295
BPL 86
BRA 119
Branch 44, 81, 82, 202, 237, 297
Break 110, 259
Break-Flag 60
Breakpoint 316, 317, 319, 320
Breakpoint
-, bedingt 320, 321
-, unbedingt 0, 320, 321
BRK 38, 52, 109, 110
BSOUT 100, 158, 164
Bug 38
BVC 88

BVS 88
.byte 215
Byte 20, 170
Bytetabelle 215, 280
Bytewert 215

C-128-Bordmonitor 130
C-128-Modus 138
C-64-Modus 138
C-Flag 164, 324
Carriage-Return 160
Carry-Flag 56, 71, 84, 89, 164
Carrybit 98
.case of 247
Case-Konstruktion 247
.caseend 247
Cdlins 272
.chain 224
CHAIN 328
Chain-Verkettung 224, 230
.chained 224
CHAINEND 328
Character-ROM 329
CHKIN 164
CHRGET 68, 92
chrin 155
CIA 130
CKOUT 164
CLC 56
CLD 60, 95
CLI 58, 95
.clist 270
CLOSE 164
CLRCH 164
CLV 61
CMOS-Prozessor 325
CMP 89
CMPARE 147
cmpvec 147
.code 204
Codeverschiebung 154
Commodore-Taste 314
Common 226, 229
-, Area 131, 138, 204
Compiler 16
.cond 254
.continued 224
CONTINUED 328
.control 256
CP/M-Modus 138
CPU (Central Processing Unit) 15
CPU-Maskenfehler 111
CPX 89
CPY 89
Cursorort 155, 157

Datazeilen 170
Dateiname 162, 163
Datei schließen 162
Dateiverwaltung 161
Datenbus 21, 22, 29
Datenrichtungsregister 132
DBM 284, 303, 304, 305
DBM-Befehle 358
DCP 112
Debugger 316, 317
Debugging 38
DEC 68
.define 210
Deklaration 292
Dekrement 65
dekrementieren 68
DEX 65
DEY 65
Dezimal-Flag 59, 75, 94
Dezimalsystem 20
Dezimalzahl 189
Dialogstring 212
DIN
-, 66001 32
-, 66261 32
Directory 184, 313
Directory-Sektor 314
Direkt-Assemblierung 308
Direktmodus 180
Disassembler 171
Disassemblierung 41, 307, 308
Disk-Monitor 313
Diskbetrieb 312
Displacement 82
Division 190
.do 243
Do-Schleife 243
Dokumentation 35, 269
DOP 112
DOS-Support 184, 312
Drucker 314
Drucker-Interface 181, 270
Druckkanal 314
Dualsystem 15, 18, 20
Dumps, der Symboltabelle 194
Durchnumerieren 183
.during 240, 242

.e
Ecklabel 329
.econd 254
Editor 167
Editor-Befehl 180
Editortypen 168
Einer-Komplement 72, 79, 80, 80
Einfügen von Zeilen 183
Eingabe, eines Zeichens 155
-, vom Bildschirm 160
-, von Datei 162
-, von der Tastatur 160
Einheit 23
Einsprungpunkt 290
.else 245
.end 210
.enif 245
Entries 289, 290
EOR 80, 96
Ersetzen, von Befehlen 187
Erweiterungs-ROM 133
exit 197
Exklusiv-ODER 76
Exklusiv-ODER-Verknüpfung 80, 96
exklusives Oder 191
.exloop 243
EXOR 76
Expansion-Port 130
.extern 291

Fallunterscheidung 239, 244, 254
False 76
Farb-RAM 130, 135
farvec 148
fast 304
Fehlerkanal 312
Fehlermeldung 192, 296, 306, 332
Fehlernummer 163
Fehlersuche 317
Fensterbreite 156
Festlegung, der Ausgabe auf Datei 162
Festsetzung
-, der Dateinamen 162
-, der Dateiparameter 162
FETCH 144
fetvec 145
Filenummer 162
Flag 26
Fliekommaroutine 166
Formatieren 196
Full-Screen-Editor 168
Füllen eines Bereiches 311
Funktionstaste 198, 312

Gerätenummer 162, 312, 314
getin 160
GETIN 161
Goto 319
Grafikschirm 139
Grundrechenarten 94, 190

Hashkode 231
Hashverfahren 195
Hex-Dump 171, 307
Hexadezimalsystem 16
Highbyte 170, 190

Hintergrund 326
HOME 305
Hot Spot 319
Hypra-Ass 226

I/O-Port 29
IEC-Bus 270
.if 245
Immediate 43
In/Out-Bereich 129, 130, 134
INC 68
indcmp 151
Indexregister 26, 62
indfet 151
indsta 151
Inkrement 65
inkrementieren 68
Integerarithmetik 189
Integerzahl 72
Interpreter 16
Interrupt 28, 108
Interrupt-Enable-Flag 58, 110
Interrupt-Leitung 22
Interrupt-Routine 108
INX 65
INY 65
IRQ 59, 1089
ISC 112

JMP 81
JMP-Befehl, indirekter JMP-Befehl 111
jmpfar 147
JMPFAR 149
Joker 187, 315
JSR 104
jsrfar 147
JSRFAR 148
Junktion 76
Junktor 76

Kernal-Editor 130
Kernal-ROM 130, 133
KIL 112
Kilobyte 20
Klammer 192
Kode
-, relokatible Kode 174
-, selbstmodifizierender Kode 145
Kodeablageadresse 204
Kodeverschiebung 204
Kommentar 177
Konfiguration 129, 305
Konfigurationsindex 129, 142
Konfigurationsregister 130, 131, 133
Kontrollausdruck 256
Kopfzeile 270
kopieren 168
Kopieren von Modulen 277

Label 172, 177, 208, 256, 261, 329
-, drucken 331
-, externe Label 173
-, globale Label 210
-, interne Label 173
-, unbekannte Label 207
Labelfile 329, 331
Labelredefinition 213
Laden 184, 312
-, absolutes Laden 164
-, der Datei 162
-, relatives Laden 165
LAR 112
LAX 112
LCR 135
LDA 50
LDX 50
LDY 50
Leerzeichen, 186, 188
LIFO 25
Line-Assemblierung 308
.link 286
Linker 167, 174, 286
.list 270
Listen 182
LOAD 51, 162, 164
Load-Configuration-Register 135
Loader 174, 281, 282
logisches Und/Oder 191
Lokaler Block 278
Loop 243
Löschen
-, von Zeilen 165, 181
-, des gesamten Textes 175, 182
Lowbyte 170, 190
LSR 98

M-Befehl 40
.macend 261
.maclib 265
Makro 167, 173, 259
Makroassembler 167
Makroaufruf 261
-, rekursiv 264, 267
Makrobibliothek 264
Makrodefinition 173
Makroname 173, 195
Makroparameter 263, 292
.malins 272
Marke 172
Maschinenkode 16, 203, 327
Maschinenprogramm 153
Maschinensprache 15
Masterdiskette 168
Meldung 257
Memory Management Unit (MMU) 128
Memory-Dumps 307
merge 184
Mergen von Quelltexten 185
Minimac 232
Minimacparameter 293
MMU 22, 130
MMU, Register der MMU 131
MMU-Version 141
Mnemonic 15, 40, 171, 177, 309
Mode-Configuration-Register 137
.modul 31, 274
Modul, relokatibel 273
Modulerzeugung 292
Modulkopf 279, 287
Modulrumpf 278, 279, 287
Monitor 38
Monitorstart 303
Multiplikation 190

Namensstack 231
Nassi-Shneidermann 32
Negation 79
-, (Einerkomplement) 191
Negativ-Flag 62, 86, 89
Neustart 223
Nibble 20
NICHT 76, 79
NMI 28
.nonum 272
NOP 106, 107, 113
Nops 272
NOT 79

.object 205
Objektcode 16, 205
ODER 76
ODER-Verknüpfung 78, 96
.of 247
Öffnen, der Datei 162
Offset 82, 323
Old 195
Opcode 40, 170, 315, 337
-, illegal 112, 171, 268, 342
OPEN 163
Operation 76, 191
Operationskode 309
Operator 190, 192
ORA 96
Ordnungszahl 278
.otherwise 247
Overflow 30, 88
-, Flag 61

Page 182, 271
Pagepointer 140
Parameterliste 260
Pass 173, 255
PC 26, 170, 191
PCR 135
PHA 101
PHP 56
PHX 119
PHY 119
PLA 101
PLOT 157
PLP 56
PLX 119
PLY 119
Position
-, der Mnemonics 182
-, des Gleichheitszeichens 182
-, des Kommentars 182
Pre-Configuration-Register 131, 135
PRIMM 158
.print 257
print ds$ 184
Programm
-, residentes Programm 38
-, transientes Programm 38
Programmabbruch 326
Programmierumgebung 16
Programmierung
-, modular 289
-, strukturiert 178, 248
Programmstart 326
Programmzähler 26, 81, 104, 170, 191, 202
Prozessor 327
-, 6502 15, 21, 29
-, 65C02 15, 119, 325
-, 65SC02 119, 325
-, 8502 15, 21, 29
Prozessor
-, Architektur 21
-, Befehle 337, 342
-, Flag 349
-, Port 132, 135
-, Register 323
-, Stack 132
-, Statusregister 56
Pseudobefehle 355
Pseudoop 172
.public 291
Pufferspeicher 153

Quelltext 16, 167, 172, 222, 273, 284, 328
Quelltext-Eingabeformat 176
Quelltextspeicher 196
Quicksort 195, 248, 253

RAM-Bank 128, 129, 133, 204
RAM-Configuration-Register 138
RAM-Makro 260
REASS 325
Reassembler 325
Rechnen im Editor 189
Referenz, extern 284, 289, 290, 292, 294
Register 318
Registeranzeige 317
Relozierung 279
.rename 184
.renumber 183
.request 212
.repeat 241
Repeatschleife 240
RESET 28
Reset 41
RESTORE 223
.revers 221
RLA 113
RMB 119
ROL 98
ROM-Erweiterung
-, intern/extern 130
ROM-Listing 166
ROM-Routine 281, 293
ROR 98
Rotieroperation 98
RRA 113
RS232-Puffer 154
RTI 61, 109, 110
RTS 104
Rücksprungadresse 104
run 197
RUN/STOP 223

SAVE 164
save 165
SBC 94
Schachtelungstiefe 211, 264
Schaltung, der Eingabe auf Datei 162
Schiebeoperation 56
Schleifenkonstruktion 239
.scratch 184
.screen 221
SCRORG 156
SEC 56
SED 60, 95
Sedezimalsystem 16, 20
Sedezimalzahl 189
SEI 58, 95
Seitenlänge 271
Sektor 313
Sekundäradresse 162, 163
Selbstmodifikation 145
.set 215

SETBNK 162, 162
SETNAM 163
SETPAR 163
Sicherheitskopie 169
.skip 271
SLO 113
SMB 119
Software-Interface 271
Sound-Generator SID 130
Source 222, 264
Sourcecode 16
Space 93
.space of 220
Spaltennummer 157
Speicher reservieren 220
Speicherbank 22
Speicherinhalt 308
Speicherkonfiguration 129
Speichern 184
Speicherbelegung 196
Speicherverwaltung 22, 128
Speicherzellen 22
Splitscreens 135
Spritedefinition 154
Sprung 82
Sprungbefehl 8
Sprungverteiler 217, 218, 248
SRE 113, 113
STA 50
Stack 24, 25, 101
Stackpointer 24, 101, 102
Startadresse 198
STASH 146
status=$90 164
Statusregister 26, 55
Stern "*" 191
Steuerbus 22, 28
Steuerkode 271
STORE-Befehl 53
String 30
Struktogramm 32
Struktur 238, 297
STX 50
STY 50
STZ 119
Subroutine 81, 104
Subtraktion 190
Subtraktion, binäre Subtraktion 71
Suchen 315
-, und Ersetzen im Quelltext 185
-, und Ersetzen von ASCII-Strings 188
-, und Ersetzen von Befehlen 186
Suchstring 187
Symboltabelle 173, 173, 213, 269, 272

.syntax 206
Syntaxfehler 175
Systemtakt 28, 28

Tabulator 182
Taktfrequenz 196
Tastaturabfrage 258
TAX 63
TAY 64
Tedmon 276
Terminologie 169
Testlauf 205
Text in reversem Bildkode 215
Textschirm 139
Token 179, 186
TOP 113
TRB 119
Trennzeile 315
True 76
TSB 119
TSX 101, 102
TXA 63
TXS 101, 102
TYA 64

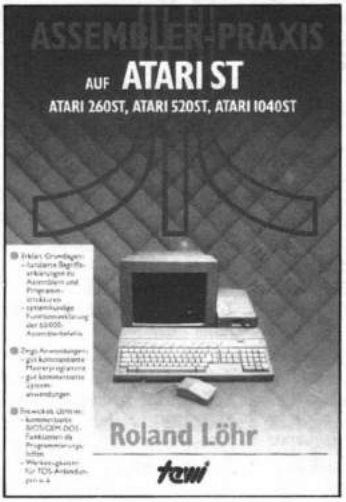
Überlauf 88
Überlaufflag 61, 88, 107
Übertrag 71, 73, 75
UND-Verknüpfung 76, 77, 96, 107
.unless 240, 242
Unterprogrammaufruf 104
.until 241
Userbreakpoint 320, 323
Userprogramm 323
Userstack 253

Variable 193, 208,
-, globale Variable 210
-, lokale Variable 194, 209
-, unbekannte Variable 207
Variablen-Statusbyte 231
Variablendeskriptor 231, 278
Variablenfeld 231, 282
Variablenname 193, 231, 278
Variablenraum 195, 231
Variablenstatus 279
Variablenwert 231, 278
Vektor 108, 108
-, IRQ-Vektor 108
-, NMI-Vektor 108
-, RST-Vektor 108
Vergleichsbefehl 89
Verifizieren 312
Verify 165
Verknüpfung, logische
Verknüpfung 96
Verschiebung, der Kodespeicherung 203
Version 141
Versionsregister 141
VIC-Chip 139
Videocontroller 130
VLO 281, 284
Vorzeichenbit 72

Wahrheitstafel 76
.wait 258
Wertzuweisung 210, 212
.while 243
WINDOW 156
Word 20
Workspace 306, 309
Wort 170

Z80-Prozessor 138
Zahl
-, Invertierung einer Zahl 79
-, sedezimale Zahl 189
Zahlenbereich, der Integerarithmetik 192
Zahlenformat 189
Zahlensystem 39, 305
Zeichen 194
Zeichenfarbe 326
Zeichensatz-ROM 130, 134
Zeile 207
Zeilen-Assemblierung 308
Zeileneditor 168
Zeilennumerierung 157, 176, 180, 328
Zeilenvorschub 315, 315
Zeroflag 89
Zeropage 26, 132, 152
Zweier-Komplement 73
Zwischenkode 174, 274

Bücher für ATARI-ST-Computer



R. Löhr

Assembler-Praxis auf ATARI ST

1986, 312 Seiten

Die Assembler-Programmierung ist eine der bedeutendsten Programmierarten, die bekanntlich den am schnellsten ausführenden Code erzeugt. Dieses Buch geht von geringen Grundkenntnissen aus und erklärt zunächst die Grundelemente der Assembler-Sprache und der Assemblierung. Anhand gut kommentierter Musterprogramme, Befehlsfunktionen, Systemanwendungen, BIOS/GEM-DOS-Funktionen als Programmierungshilfen, TOS-Anbindungen und weiterer Beispiele wird der Anwender in die professionelle Assembler-Programmierung auf ATARI ST eingeführt.

- Eine systemkundige Assembler-Darstellung.

Best.-Nr. 80370
ISBN 3-921803-70-5
DM 59,–/sFr 54,30/öS 460,20

I. Lüke/P. Lüke

Der ATARI 520 ST

2. überarbeitete und erweiterte Auflage 1986, 198 Seiten

Dieses Buch enthält alle Informationen, die für Interessierte und für alle stolzen Besitzer eines gerade erworbenen ATARI 520/260 ST wichtig sind. Die jetzt vorliegende überarbeitete und erweiterte Auflage trägt den neuesten Entwicklungen bei ATARI Rechnung. Unter anderem wurden das inzwischen deutschsprachige Betriebssystem und einige geänderte Systemausstattungsmerkmale berücksichtigt. Das Buch ist somit nicht nur eine Rechnerbeschreibung mit hohem Informationswert, es leistet auch als Nachschlagewerk wertvolle Dienste.

Best.-Nr. 90229
ISBN 3-89090-229-4
DM 49,–/sFr 45,10/öS 382,20

A. Steiner/G. Steiner

GEM für den ATARI 520 ST

2. überarbeitete und erweiterte Auflage 1986, 334 Seiten

Die Benutzeroberfläche des neuen ATARI ST – GEM genannt – erhebt den Anspruch, die Bedienung des Computers zum Kinderspiel zu machen. Dennoch: Wenn Sie die bisher übliche kommandoorientierte Umgangsweise mit Ihrem Computer pflegten, so werden Sie eine Einführung in die Bedienung von Maus, Bildsymbolen und Fenster, wie sie dieses Buch liefert, zu schätzen wissen. Besonders interessant für den erfahrenen Anwender sind die Kapitel über den internen Aufbau von GEM mit seinen Pull-Down-Menüs, Fenstern und Symbolen.

Best.-Nr. 90230
ISBN 3-89090-230-8
DM 52,–/sFr 47,80/öS 405,60

Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computerfachgeschäften oder bei Ihrem Buchhändler.

701311

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bücher für ATARI-ST-Computer

P. Rosenbeck

C-Programmierung unter TOS/ATARI ST
1986, 376 Seiten

Erst durch das Programmieren in C kann der stolze Besitzer alle Fähigkeiten seines ATARI ST ausnutzen. Für Leser mit elementaren EDV-Vorkenntnissen gibt der Autor in diesem Buch eine gründliche und leicht lesbare Einführung in das Programmieren mit dieser wichtigen und vielseitigen Sprache. An aussagekräftigen und in allen Einzelheiten erklärten Beispielen werden auch die fortgeschrittenen Aspekte der Sprache besprochen.
Best.-Nr. 90226
ISBN 3-89090-226-X
DM 52,–/sFr 47,80/öS 405,60

J. Purdum/T. Leslie

Die C-Programmbibliothek
1986, 361 Seiten

Dieses Buch erspart dem C-Programmierer Stunden mühseliger Kleinarbeit und hilft, effizientere Programme zu schreiben. Es ist in zwei Teile gegliedert. Der erste Teil zeigt, wie man zu universellen Bibliotheksfunktionen kommt, und gibt Tips, wie C noch wirkungsvoller eingesetzt werden kann. Der zweite Teil enthält eine Reihe ausführlich erklärter C-Funktionen als wertvolle Ergänzung Ihrer Programmbibliothek. Dazu gehören unter anderem ein Terminalinstallationsprogramm, mehrere Sortier-Algorithmen und ein Satz ISAM-Funktionen.
Best.-Nr. 90133
ISBN 3-89090-133-6
DM 69,–/sFr 63,50/öS 538,20

W. Hilf/A. Nausch

M68000-Familie: Teil 1
1984, 568 Seiten

Informative Einführung in die Geschichte und die Entwicklungsphilosophie einer detaillierten Darstellung der Hardware sowie ausführliche Erläuterung der komfortablen Adressierungsarten.
Best.-Nr. 80316
ISBN 3-921803-16-0
DM 79,–/sFr 72,80/öS 616,20

M68000-Familie: Teil 2
1985, 400 Seiten

Teil 2 des umfassenden Lehr- und Nachschlagewerks zum M68000 beschäftigt sich mit Anwendungen und weiteren Mitgliedern der M68000-Familie.
Best.-Nr. 80330
ISBN 3-921803-30-6
DM 69,–/sFr 63,50/öS 538,20

Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computerfachgeschäften oder bei Ihrem Buchhändler.

701313

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Spitzen-Software für ATARI-ST-Computer

WordStar 3.0 mit MailMerge

Der Bestseller unter den Textverarbeitungsprogrammen bietet Ihnen bildschirmorientierte Formatierung, deutschen Zeichensatz und DIN-Tastatur sowie integrierte Hilfstexte. Mit MailMerge können Sie Serienbriefe mit persönlicher Anrede an eine beliebige Anzahl von Adressen schreiben und auch die Adreßaufkleber drucken.

Jetzt gibt es WordStar/MailMerge für den ATARI ST! Damit eröffnen sich Ihnen alle Möglichkeiten, Ihren ATARI ST für professionelle Textverarbeitung einzusetzen. Zum Superpreis!

WordStar für die ATARI-ST-Computer wird auf zwei 3½-Zoll-Disketten geliefert. Sie beinhalten:

- CP/M-Z80-Emulator
- WordStar/MailMerge-Dateien

Hardware-Anforderungen: ATARI-ST-Computer, 80-Zeichen-Monitor, ein 3½-Zoll-Diskettenlaufwerk, beliebiger Drucker mit Centronics-Schnittstelle.
WordStar ist an den ATARI ST bereits fertig angepaßt und läßt sich bequem über Funktionstasten steuern.

Bestell-Nr. MS 106
Für sagenhafte DM 199,– *
(sFr 178,–/öS 1890,–*)
*** inkl. MwSt. Unverbindliche Preisempfehlung**

Und dazu die weiterführende Literatur:

WordStar für ATARI ST

Mit diesem Buch haben Sie eine wertvolle Ergänzung zum WordStar-Handbuch: Anhand vieler Beispiele steigen Sie mühelos in die Praxis der Textverarbeitung mit **WordStar** ein. Angefangen beim einfachen Brief bis hin zur umfangreichen Manuskripterstellung zeigt Ihnen dieses Buch auch, wie Sie mit Hilfe von MailMerge Serienbriefe an eine beliebige Anzahl von Adressen mit persönlicher Anrede senden können.

Best.-Nr. 90208
ISBN 3-89090-208-1

DM 49,– (sFr 45,10/öS 382,20)

Erhältlich bei Ihrem Buchhändler.

Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computerfachgeschäften oder bei Ihrem Buchhändler.

701314

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bücher für ATARI-ST-Computer

R. Aumiller/D. Luda

Programmieren mit Forth Atari ST

1987, ca. 250 Seiten
inkl. Diskette

Die Einführung in Forth erfolgt schrittweise über einfache Programme bis hin zur fortgeschrittenen Programmierung. Dem Profi zeigt eine ausführliche Darstellung der Programmierung von Forth unter GEM, wie vielfältig die Fähigkeiten des Atari ST genutzt werden können. Alle GEM-Routinen werden ausführlich beschrieben, und es wird gezeigt, wie man sie einsetzen kann. Dem Buch liegt eine Diskette mit über 50 Beispielprogrammen bei.

Best.-Nr. 90237
ISBN 3-89090-237-5
DM 49,–/sFr 45,10/öS 382,20

W. F. Fastenrath

ATARI-ST-BASIC-Handbuch

1986, 264 Seiten

Das BASIC für die ATARI-ST-Computer ist außerordentlich umfangreich und mächtig. Über 130 Befehle stehen bereit, um auch komplexere Aufgaben mit diesen Computern zu bewältigen.
Dieses Buch beabsichtigt nicht eine allgemeine Einführung in die Techniken der BASIC-Programmierung. Es stellt vielmehr eine Anleitung zur Anwendung von BASIC auf die Erfordernisse und Möglichkeiten dieses speziellen Systems dar. Eine übersichtliche Zusammenstellung des gesamten Befehlsvorrats macht dieses Buch zu einem Hilfsbuch bei der täglichen Programmierarbeit.

Best.-Nr. 90205
ISBN 3-89090-205-7
DM 52,–/sFr 47,80/öS 405,60

R. Aumiller/D. Luda

ATARI-ST-LOGO

1986, 236 Seiten

LOGO – einfach wie BASIC, jedoch so leistungsstark wie Pascal oder FORTH. LOGO vereinigt viele Vorteile anderer Programmiersprachen in sich. Es ist interaktiv, listen- und prozedurorientiert, erweiterbar, einfach zu erlernen und doch komplexen Problemen gewachsen.
Dieses Buch ist für Anfänger und Fortgeschrittene gleichermaßen geeignet. Bildschirmfotos, viele ausführliche Beispiele – teilweise mit Übungsaufgaben zur Vertiefung des Gelesenen – tragen zu einer guten Verständlichkeit und einem sicheren Lernerfolg bei.

Best.-Nr. 90223
ISBN 3-89090-223-5
DM 49,–/sFr 45,10/öS 382,20

Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computerfachgeschäften oder bei Ihrem Buchhändler.

701312

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

ATARI-ST-ASSEMBLER-BUCH

PETER WOLLSCHLÄGER, Jahrgang 1939, gehört noch zu jener Technikergeneration, die digitale Schaltungen bis hin zum Computer mit einzelnen Transistoren entwickelte. Programmiert hat er damals nur nebenbei. Mit Einführung der Mikroprozessoren verlagerte sich seine Tätigkeit immer mehr von der Hardware zur Software. Durch ständige Weiterbildung, zahlreiche Kurse und Workshops bei den Herstellern sowie einige nachgeholte Semester in Informatik und über 20 Jahre praktische Erfahrung als Systemprogrammierer wurde er zum Experten für Mikrocomputer. Nebenberuflich arbeitet er als freier Autor für »Computer persönlich« und »mc«. Die journalistische Erfahrung aus über 150 Artikeln, immer unter der Prämisse geschrieben, schwierige Themen gut verständlich und angenehm lesbar darzustellen, steckt in diesem Buch.

Jede höhere Programmiersprache wie Basic oder Pascal, aber auch noch C, legt dem Anwender Beschränkungen auf. Sei es, daß bestimmte Dinge sich partout nicht realisieren lassen oder sei es, daß Programme viel zu langsam laufen. Der logische Entschluß, dann tiefer einzusteigen, erhält auch prompt seinen Dämpfer. Was nun käme, sei Assembler, und das sei wirklich nur etwas für Profis. Erst recht gälte dies für so einen komplizierten Prozessor wie den 68000.

Mit diesem Buch soll bewiesen werden, daß Assembler auch nur mit Wasser gekocht wird. Es ist nämlich ganz einfach. Wer eine Programmiersprache gelernt hat und sie erfolgreich anwendet, der lernt auch andere. Assembler macht da keine Ausnahme.

Allerdings setzt Assembler ein gehöriges Maß an Grundwissen über computerinterne Dinge voraus. Aber keine Angst – nach einem Minimum an Theorie geht es sofort in die Praxis. Assembler-Befehle und TOS-Funktionen werden anhand kleiner Programme erklärt. Die Programme werden ständig schwieriger, das erforderliche Wissen wird dabei von Fall zu Fall mitgeliefert.

Mit einem allgemein verwendbaren Rahmenprogramm, einer RAM-DISK, einem Diskettenmonitor und vielen anderen Utilities bleibt schließlich für den Leser vieles, was er auch später immer wieder verwenden kann.

Aus dem Inhalt:

- Grundlagen des 68000
- Systemprogrammierung anhand vieler Beispiele
- Superschnelle Grafik
- Alle GEMDOS-, BIOS- und XBIOS-Funktionen
- Das File-System des ST
- Die Tricks der Profis

Dem Buch liegt eine 3½"-Diskette mit allen Beispielprogrammen bei.

Hardware-Anforderung:
Atari-ST-Computer
(260 ST, 520 ST, 1040 ST),
Diskettenlaufwerk und Monitor

Software-Anforderung:
Assembler-System für Atari ST

ISB N 3-89090-467-X

4 001057 904675

Markt&Technik

DM 59,-
sFr 54,30
öS 460,20

Stichwortverzeichnis

2er-Komplement 73

Absolute Adressierung 46
Adresse 18, 28
Adresse
–, effektiv 57, 61
–, logisch 28
–, physikalisch 28
–, symbolisch 80
Adressierung 36, 42, 44
Adreßbus 28
Adreßregister 40, 41
–, indirekt 45
AES 93
ALU 181
Animation 159, 162
Arbeits-Register 117
ARI 45
–, mit Adreßdistanz 45
–, mit Adreßdistanz und Index 46
–, mit Postinkrement 45
–, mit Predekrement 45
Arithmetische Befehle 172
ASCII-Code 53, 74
ASCII-Display 206
Assembler 18, 21
Assembler-Basic-Konverter 222
Assembler-Direktive 56, 85
Assembler-Listing 102
assemblieren 54
Attribut 122
Ausgabe
–, von Strings 56, 114
–, von Zeichen 52

Base-Page 146
Basic 221
Basis-Adresse 47
Batch 106
BATCH.TTP 55
BCD (Binary Coded Decimal) 40
BCD-Arithmetik 173
Befehl 18, 28, 51
–, Aufbau 41
–, Decoder 181
–, Emulatoren 151
–, Liste 241
–, Register 181
–, Wort 28, 44
Benutzeroberfläche 23
Bildschirm-Gestaltung 215
binär 32
Binärzahlen 32, 44
Binder 22
BIOS 89
BIOS-Funktionen 91, 265
BIOS-Parameter-Block 129
Bit 17
Bit-Befehl 175
Bit-Plan 158
Bit-Schieben 73
Blockzuweisungs-Tabelle 132
Bomben 33, 294
Boot-Sektor 121, 129, 190
BPB 130
Branch-Befehle 72
BSS 64, 146
Bug 22
Bus 27, 28
Bus-Error 28, 183
Byte 17, 40

Carry-Flag 76
CASE X OF 82
Cartridge-Programm 230
CCR (Condition Code Register) 59, 71
Chip 28
Cluster 121, 132
Cluster-Liste 136
Code-Module 105
COMMAND.PRG 93
compare 59
Compiler 19
Condition Codes 254
Control-C 64
CPM/M-68K 89
CPU 17, 18, 27
CR 64
Cross-Assembler 103

Data 64
Date-Segment 146

Daten 18, 28, 121
Datenbereich 121
Datenregister 40, 41
Datensegment 65
Datensicherungsbereich 237
Datenwort 29
Datum 123
DBcc 62
DBcc-Schleife 61
Debugger 22
–, symbolisch 145
define storage 64
DFB 62
Directory 121
Directory-Entries 122
Direktive 64
Disassembler 152
Diskeditor 205
Diskmonitor 205
Dispatcher 51
Displays 215
DIVS 110
DIVU 110
DR-Format 23
Drive 191
ds 64
DTA-Puffer 124
Dualzahl 31, 32

ea 44
Editor 21
Eingabe von Strings 63
Endlosschleife 64
Entwicklungspaket 23
EPROM 233
equ 85
EQU-Direktiven 126
Equate 85
EVEN 29, 65
Exception 29, 151, 183
Exception-Vektoren 30, 293
External 105

Fastlink 23, 54
FAT 1 121, 131
FAT 2 121
Fehlercode 66
Fehlermeldung 21, 263
Fehlernummer 143
Fetch 28
File 121
File Allocation Table 132
File-Größe 123
File-Handling 142
File-System 121
Flags 59, 72
Force Uppercase 84
Formatieren 216
Funktionsnummer 51
Funktions-Taste 77

GEM 93
GEMDOS 53, 57, 89
GEMDOS-Funktionen 90, 255
Global 105
Grafik-Kern 89
Grafik-Routinen 277
GST-Assembler 24, 99
GST-Format 23

Halbbyte 40
Hard-Disk 191
Hdv-Vektoren 190
Header 22, 145, 233
Hex-Code 45
Hex-Darstellung 74
Hex-Dezi-Konvertierung 31
Hex-Display 206
Hex-Dump 144
Hex-Konverter 74
Hex-String 73
hexadezimales Zahlensystem 31
Hochsprache 19, 20

IDEAL 24
IF THEN 71
in memory 21
include 104, 105
Include-Files 21, 104
Interpreter 19
Interrupt 182

K-SEKA 23
Kaltstart 182
Kill Return Address 82
Kommentar 51
Konstante 46, 53
Konstanten-Adressierung 46
Kopier-Programm 201
Kopierschutz 216

Label 22, 80
LC 85
LF 64
Library 104
LIFO 33
Line-A-Grafik 154, 159, 277
Line-A-Handler 152
Line-A-Variablen 155
Line-F-Emulator 154
LINK 171
Linken 20, 54
Linker 22
List-Option 102
Location Counter 85, 102, 143
Logische Befehle 174
Low-Level-Routinen 139

Mac-Lib 104
Makrodefinition 104
Makrofähig 21
Makros 21, 99, 100, 127, 142
Makro-Sprache 99
Marke 22, 51
Maschinensprache 17, 45, 102
Maske 75
Megamax-Assembler 225
Mehrfachverzweigung 78
Menü-Technik 84, 214
Metacomco-Assembler 23
Mikro-Code 181
MMU 28
Modul 104, 221

Nano-Code 182
Nibble 40, 75
Nullbyte 67

Objekt-Code 21, 102
Objekt-File 21, 54
ON X GOSUB 79
Operand 44
Operanden 41, 51
OS (Operating System) 51

Parameter 65
Parameterübergabe 55
Pascal 19, 223
patch 191
PC 28, 29, 151
PC-relativ 47
pea (Push Effective Address) 61
Peripherie 18
Pilze 33
POKEn 221
Präfix 33
Prefetch 182
Priorität 184
Profimat 24
Programm 17
–, Counter 28
–, Entwicklung 108
–, Kopf 144
–, Steuerbefehl 176
Prozessor 27

Quell- und Zieloperand 44
Quelle 44
Quelltext 22

RAM 27
RAM-Disk 189
–, vollautomatisch 200
Rechenwerk 181
Register 39
–, direkt 45
Register-Modell 39
Register-Operation 40
Relokatier-Tabelle 145
Reset 28
Return-Adresse 55
ROM 27, 233
Rotierbefehl 175
RTE 153

Scannen 76
Scan-Code 76, 78, 80
Schiebebefehl 175
Schleife 59, 62
Schleifenbefehl 62
Segment 64
Sektor 121
–, relativ 121
Shell 22
Side 121
SOURCE STATEMENT 102
Source-File 21
SP 33
Speicher 28
Speicherbedarf 190
Sprite 166
Sprite-Definition 166
Sprite-Editor 166
Sprungtabelle 80, 81
ST starten 229
ST-Pascal 54
Stack 33, 34
Stackpointer 33, 40
Stapelspeicher 33
Stapelzeiger 33
Start 229
Start-Cluster 122
Status 143
Statusregister 40, 41, 71, 151
String 56, 58
–, Ausgabe 59
–, Eingabe mit GEMDOS 10, 66
–, Format 66
Struktur 98
Sub-Directory 122
Suchen 86
Super-Statuswort 184
Supervisor-Bit 236
Supervisor-Modus 71, 151, 182
Symbol-Tabelle 22, 145
System- und User-Byte 71
System-Datei 123
Systemvariable 289

T.L.D.U. 205
Tabelle 82, 136
Text 64
Texteingabe 20
Textmodul 105

Top Down 107
TOS 51
TPA 146
Trace-Bit 183
Tracks 121
Transfer-Befehle 171
Trap 51
–, Befehl 182
–, Dispatcher des BIOS 234
–, Handler 153
–, Vektoren 30
TTP 147
Typ-Angabe 41
Typ-Anpassung 57
type casting 57
Typwandlung 57

Uhrzeit 123
UNLK 171
Unterprogramm 65
Usermodus 182
User-Mode 71

Variable, lokal 103
VDI 93
Verbiegen eines Vektors 185
Verschiebe-Tabelle 145
Versteckte Datei 123
VT52-Emulator 214, 285

Warmstart 182
Warnung 21
Wortgrenze 29, 65

XBIOS 89, 215
XBIOS-Funktionen 92, 269
XOR-Modus 161
XREF 105

Zahlenwandlung 109, 217
Zeichenkonstante 53
Zeichensatz 281
Zeiger 90
Zeilenvorschub 76
zero terminated 58
Ziel 44

Markt&Technik
Verlag Aktiengesellschaft
Hans-Pinsel-Straße 2
8013 Haar bei München
Tel. (089) 4613-133
Telex 522052
ATARI ST
Assembler - Buch
Ein 68000er-Kurs mit vielen Beispielen
Bestell-Nr. 90467 / 10038765
!Disk einseitig bespielt!